POLITECNICO DI MILANO

Dipartimento di Elettronica, Informazione e Bioingegneria

Master of Science in Telecommunication Engineering

Implementation and Performance of a SDN Cluster-Controller Based on the OpenDayLight Framework

Author:
Esteban Hernandez

Supervisor:
Prof. Maier Guido Alberto

April 2016

Contents

1. Introduction

In the last decade the world have witnessed as Internet traffic grows in a very fast way, this is due to different factors such as More Internet Users, Proliferation of Devices and Connections, Video Services, Mobility Momentum and many others [1]. This has resulted in the increase of large data centers with the aim of processing all this information. Due to these factors computational complexity and storage have increased tremendously, therefore making the networking complexity higher.

In response to this problem of very difficult networks to handle, a set of organizations such as operators, researchers and equipment vendors got together and saw the need of innovation, that's why was formed in 2011 the Open Networking Foundation (ONF) with the objective of promoting a new network propose, Software Defined Networks (SDN).

The fundamental idea behind SDN is a network architecture where the Control Plane is separate from the Data Plane [2], this architecture opens up the possibilities of what there was before. This is because with SDN it is possible to have programmable networks where network administrators can custom the network in order to satisfy their needs. In the SDN paradigm there is a central software program called controller, which handles the behavior of the entire network. Becoming the brain of the network and, making network devices become simples forwarding devices.

This new way of managing networks leads to the need to communicate simple forwarding devices with SDN controllers and, for this was created the communications protocol called OpenFlow.

In the SDN architecture network devices only rely on a single controller, this gives the system the weaknesses of having a single point of failure. This issue can be solve by deploying a cluster of SDN controllers, which will give the system an improvement in scalability, high availability and persistence of the information.

The principal goal of this thesis is to deploy and analyze different aspects as the internal performing, messages exchanged between controllers and, the bandwidth usage of a cluster of OpenDayLight controllers. For this was created a cluster with three instances of the controller, subsequently it was connected a Mininet network with a change in the topology size. One of the objectives was also to provide a cluster with high availability (HA), for this purpose two nodes in the cluster are acting as redundancy in case of failure. In OpenDayLight they act as followers and the other node acts as the leader of the cluster. The leader is the one in charge of communicate with all the network devices, in case of failure the system will select a new leader. Network devices will be passed out to be controlled for it.

In order to maintain information persistency and HA in the cluster, there has to be a continuous exchange of inter-controllers traffic with the aim of maintaining the other nodes updated. Then with these information is built a model to predict how is the bandwidth usage (Mbps) with the variation of the topology.

1.1. Outline

This report is organized in the following way: Chapter 2 contains some background about software defined networking. Chapter 3 contains a brief introduction to the OpenDayLight controller. Chapter 4 describes how a distributed system works. Chapter 5 explains how Raft (the consensus algorithm) works and, why is so important for the deployment of the cluster. Chapter 6 describes Akka, tool used in the discovering part. Chapter 7 explains the gossip protocol. Chapter 8 describes the cluster architecture in ODL. Chapter 9 explains the configuration of the cluster and, how to check that the cluster is running properly. Chapter 10 explains the Logs of the system. Chapter 11 shows the different messages used for ODL. Chapter 12 contains the Model for the bandwidth usage. Chapter 13 concludes the thesis and, talks about future works.

2. Software Defined Networking

The idea behind SDN networks is to pull out the intelligence from the network hardware, something that has been done in other fields of technology. Right now is being using the same that was in 1999, router, switches, routing protocols, they are faster, with bigger Backplane, more throughput, Qos and some more things were added. But intelligence is basically the same.

Networking equipment have remained unchanged along the years. In SDN, network equipment have become dumber. Allowing the creation of a management system in order to make the whole system more intelligent by having control of the network architecture. Before was only a matter of the size of the pipe, in other words speed (ex, how much time does it takes to transport something from one point A to B). Now with the arrival of new applications such as real-time communications (YouTube, skype, VoIP) the really importance was the latency or jitter. Now there is the problem of real-time communications, where is used Qos to make some kind of prioritization with packages. For example making a packet VoIP more important than an FTP and so sending it first through the network.

In the past was common to say that FTP traffic had lower priority than the SIP traffic. Now that more devices are connected to the network, it may happen that under some conditions the FTP traffic could be more important than SIP traffic. The problem with the systems that are available in today days to manage Qos is that it cannot be dynamically configured this information, this must be programmed statically. Is there where SDN gets to play a very important role, traffic can be modeled and shaped dynamically depending on what is needed, basically the control plane and the data plane are separated.

Why to shape real time traffic?

Whenever a network is configured, this will be governed by its own limitation. Most of the time an application uses the entire bandwidth allocated and sometimes only part of it. But there are occasions in which another application needs to use the bandwidth that is not being used for the other application and, unfortunately it cannot be used. This is a clear example of how interesting would be if there was an infrastructure where it could be possible to prioritize traffic in real time. For example in a pipe of 1 Mb instead of sectioning the pipe for all the services by default, it would be better to modify it dynamically according to the instant needs.

2.1. SDN theory

According to [3] traditional IP networks are complex and very hard to manage. It is both difficult to configure the network according to predefined policies, and to reconfigure it to respond to faults, load, and changes. To make matters even more difficult, current networks are also vertically integrated: the control and data planes are bundled together. Software-defined networking (SDN) is an emerging paradigm that promises to change this state of affairs, by breaking vertical integration, separating the network's control logic (Control plane) from the underlying routers and switches (data plane) as is shown in Figure 2-1, promoting logical centralization of network control, and introducing the ability to program the network.

With this separation on the control plane and data plane, all the networking devices have become dumber, they began to be only forwarding devices. The control plane is now implemented in a centralized controller.

- Data plane: is where reside all the switches and routers (forwarding devices) that allow a package to go from a point A to a point b.
- Control plane: is where reside a set of management servers which communicate with all the forwarding devices and say how data should move

in the plane data. This can be changed dynamically over the time, allowing to control the entire network from a single point. This is done by separating different components of the network infrastructure, so being able to deal with them separately.

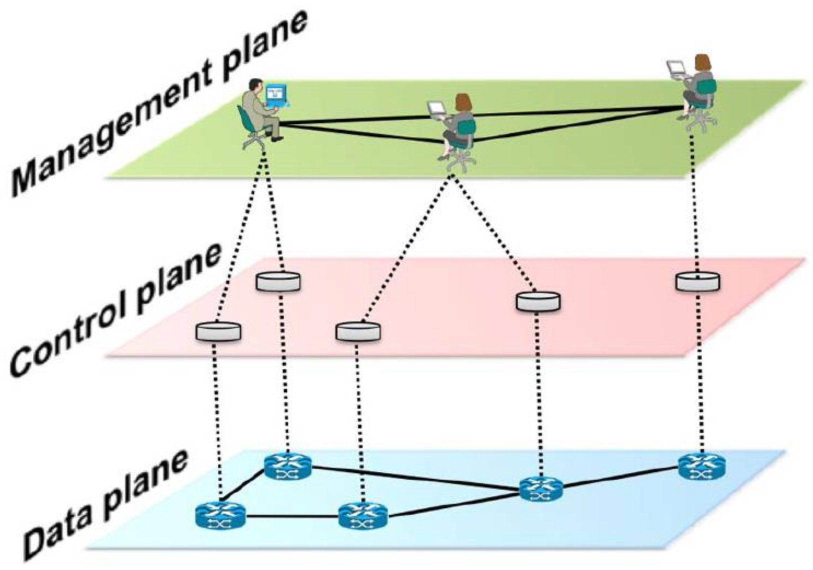


Figure 2-1 – SDN architecture *[3]*

Now that the control plane and the data plane are separated, it is needed a new form of communication for them. In SDN there is something called OpenFlow that is a protocol for controlling all the network devices.

The principle characteristics in a SDN network are: 1) control plane and the data plane are decoupled. 2) Forwarding decisions are flow based [3], a flow means a sequence of packets from a point to another. 3) Now that control plane and data plane are separated, then the logic control is moved to an external entity, In SDN this external entity is called SDN controller. 4) SDN networks are highly programmable through applications, these application are running on top of the control plane.

All the above mentioned make the SDN architecture agile, centrally managed, direct programmable and scalable [4].

2.2. SDN definition

In order to contextualize the reader, it will be explained the SDN terminology.

1) Forwarding Devices (FD): are the switches and routers that are in charge of implement all the flow rules given for the controller. They are connected to the controller through the southbound interface and, this is done by using the OpenFlow protocol.
2) Data plane (DP): is where all the forwarding devices reside.
3) Control plane (CP): is the brain of the network, it is connected with the data plane with the southbound interface and to the applications with the northbound interface.
4) Southbound interface (SI): is the way in which the controller communicate with the forwarding devices, the protocol used is the OpenFlow.
5) Northbound interface (NI): SDN architecture offer a way to program the controller and modify thing inside the controller, this is done by using this interface.

2.3. OpenFlow

This is a communication standard interface managed for the Open Networking Foundation (ONF) that is used between the control plane and the data plane in SDN network [2], basically allows the configuration of forwarding devices such switches and routers.

This protocol eliminates the problem of having static network architectures [5]. With OpenFlow is possible to create a single network control policy that can be spread through the entire network, allowing a central controller to remotely manage the forwarding information in all the forwarding devices of the data plane. This is an amazing approach because this makes the network more automatic, eliminating the problem of configure all the devices and interfaces manually one by one. Another advantage is that it won’t differ from a vendor to another, making the process easier.

2.4. Distributed Controllers

In SDN networks is also available the distributed controller architecture, the fact of distributing the control to a set of controllers gives the system the ability to have multiple points of failures [6]. Not as before that the system had a single point of failures. The implementation is shown in Figure 2-2.

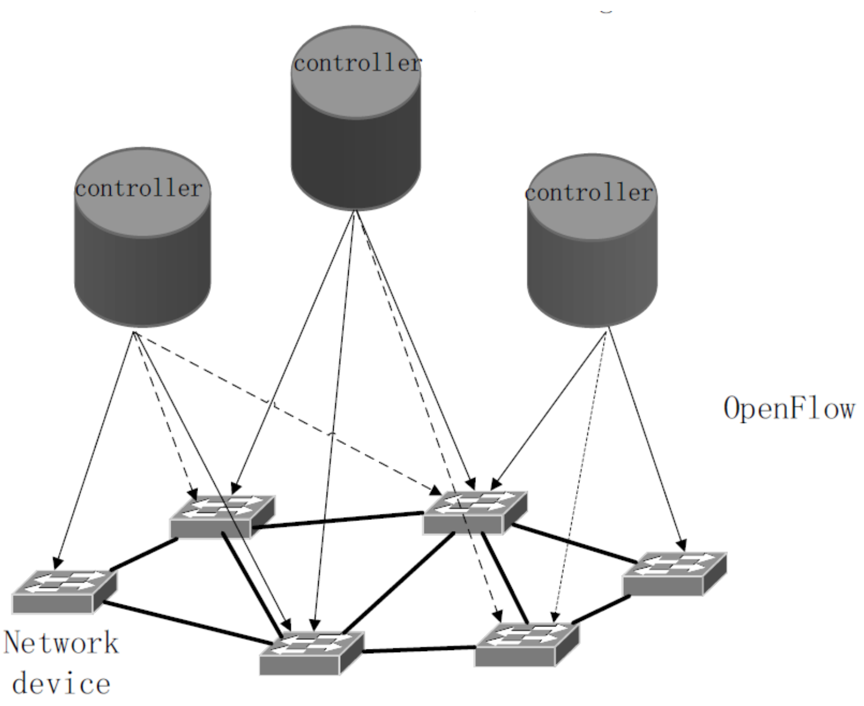


Figure 2-2 - Distributed Controllers *[6]*

Having multiple controllers running at the same time and working together also gives the network the ability to improve its scalability, persistency, share workload and, work in a high availability mode.

Controllers have to exchange some control information in order to work in a distributed way, this traffic is often called East-West traffic. In this traffic is include information about the topology network, inventory, and some other control plane parameters.

3. OpenDayLight (ODL)

OpenDayLight is a collaborative open source project that is hosted for The Linux Foundation, it was founded in April 2013 but the first release was in February 2014. It was created with the aim of reducing the known “vendor locking” and therefore supporting more protocols than only OpenFlow.

The objective of OpenDayLight is to provide a centralized management system that allows to have a programmable network. This can be achieve by using API frameworks.

OpenDayLight is a modular open SDN platform for networks of any size and scale, enabling network services across a spectrum of hardware in multivendor environments. The micro services architecture allows users to control applications, protocols and plugins, as well as to provide connections between external consumers and providers [7].

In today days all the networks have to be modified manually in order to accommodate them to the needs and workload of the moment. For this in SDN networks the OpenDayLight controller can be used as a platform for configuring different aspects of the network and solving different network challenges. ODL uses open source integration standards and APIs to make the network more programmable, intelligent and adaptable.

The controller is very adaptable to needs, this enable the ability of combine multiple services and protocols to solve different problems.

The fact that ODL is open source has been the key for the rapidly growing of it, making possible that many programmers around the world can contribute to develop software for this management system [8].

3.1. Architecture

The OpenDayLight controller has 3 different layers that are separated into: Top layer, middle layer and bottom layer as is show in Figure 3-1.

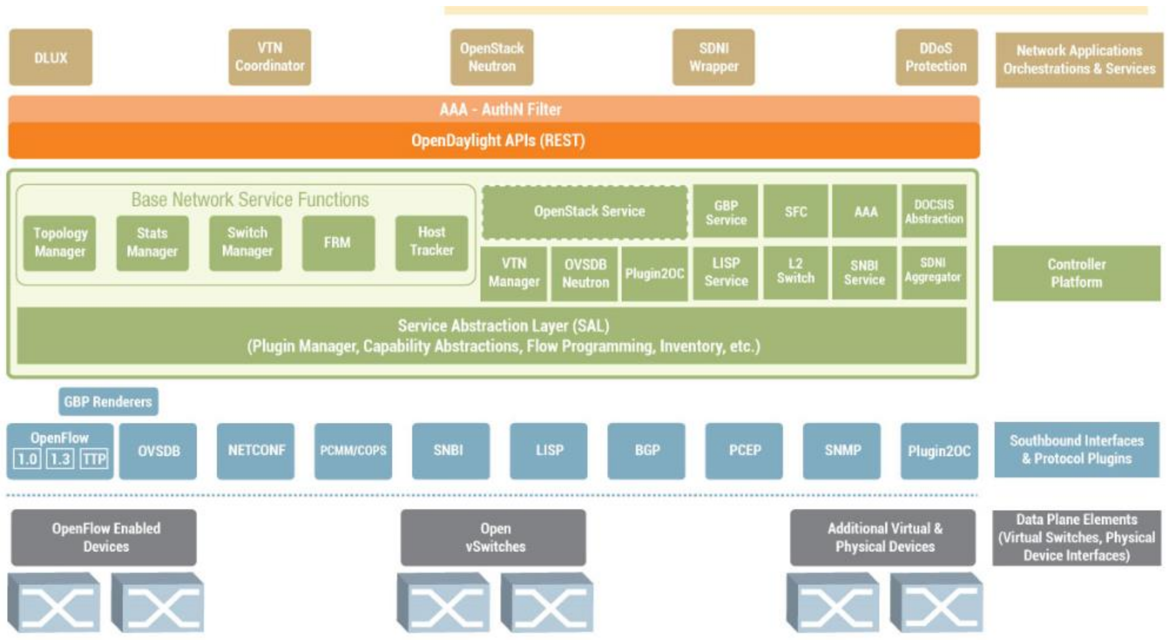


Figure 3-1 – OpenDayLight Architecture *[8]*

In the ODL architecture are included some components as a fully pluggable controller [9], interfaces, protocol plug-ins and applications. For the helium release which is the version that is being used for this thesis, will be explained the three layers of ODL.

Top layer – Northbound interface:

In this layer the northbound interface provides controller services and common REST APIs. This helps for the managing of the network infrastructure configuration.

Middle layer – Controller platform:

In this layer the controller communicates with the underlying network infrastructure with help of the southbound plug-ins. This is in charge of provide basic networking services, including topology manager and switch manager.

Bottom layer – Southbound interface:

In this layer is where all the protocols for manage and control the underlying networking infrastructure reside. It has different plug-ins that also implement various networking protocols which directly communicate with hardware. Here is where the OpenFlow protocol reside.

3.2. Services in ODL

Topology manager: is in charge of store and handle all the information about networking devices. It contain topology details as switches and links.

Statistics manager: is in charge of collecting all the statistic information, this can be done by sending statistic requests to the nodes and storing the responses. The statistic manager also communicates with northbound APIs to provide information about nodes, flows, tables and group statistic.

Switch manager: provides information about switches and ports. It can also communicates with northbound APIs to provide information.

Inventory manager: guarantees that the database of the inventory can be always as updated as possible. It queries and updates information about switches and ports managed by OpenDayLight.

4. Distributed systems

What is it?

In traditionally computation everything is performed in only one machine as is show in Figure 4-1, it does not matter if it is a computer, a mobile phone or whatever other computational device. The procedure is simple: an input is given to a computational device in order to process it and, finally obtain a desired output.

Is truth that in today days this is actually enough, but in very large scale projects, for example, while doing 3D graphics, video rendering or in fact in the even larger scale, projects, for example, if a researcher is trying to correct a complicated scientific problem. In such situations the processing power of a single computer may not be enough.

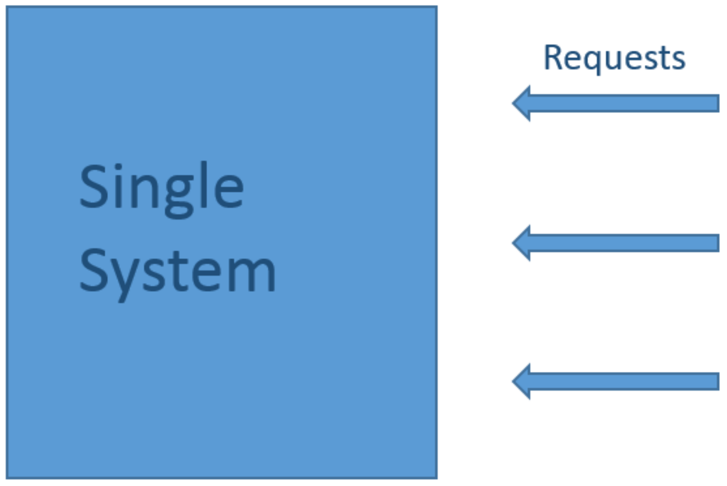


Figure 4-1 – Single System

A single computer can be maybe too slow to solve a large problem, and that’s how distributed computing comes it, the idea is pretty simple. It is taken a large complex task and it is chopped up into little bits, distributing the workload over a large number of computers so that each computer only needs to work in a small job. All the computers are supposed to work in unity and as a result a solution will be obtained in far less time than the last computation.

Taking this idea to the main work of the thesis, in which there are very large scale services, and those services are getting lots of requests from people who are trying to access them. Is there when comes out a big problem. If it is wanted to serve that amount of people at the same time with only one machine, it will be basically impossible. It cannot be built a computer that can serve anyone at the same time.

The idea is to distribute the load (number of requests) on interconnected computers that talk to each other and run the applications together as is shown in Figure 4-2. So there are computers on a network connected each other and they are handling different portions of the load. In This way is possible to scale up when the system gets more requests, just simply adding more computers to the distributed system. Another massive advantage of this approach is, for example, when there is a failure in one of the nodes, the other node will take the workload of that node. It allows to scale up, be reliable and it is an approach that make sense and is feasible.

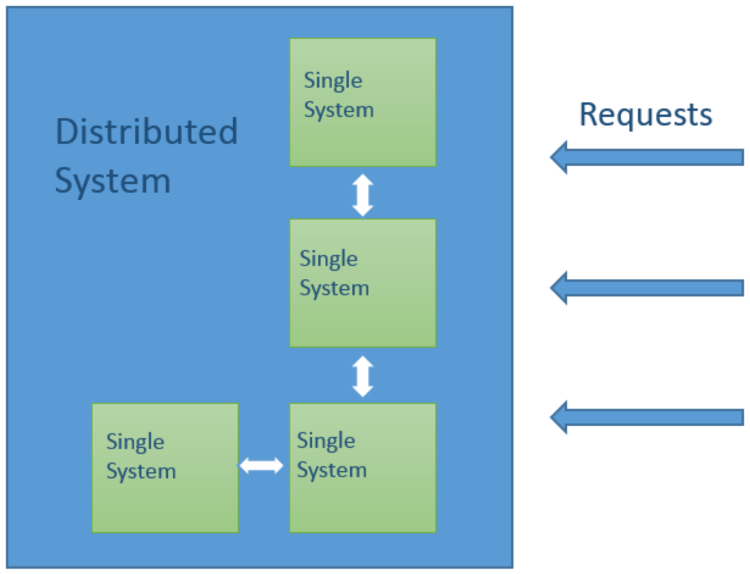


Figure 4-2 – Distributed System

The most suitable tasks for distributed systems are parallelizable tasks, it may require a large number of complicated operations but many of these operations can take place independently of each other. Which means that each task is distributed and, since one task does not rely on the result of another different task all the tasks can be done at the same time without regard of the other task.

The way this is done is simple, basically there is a host computer as well as an array of computers that are going to help with the distributed system. The host computer is where is set up the task and where is running the main program, this computer has the task of defining all the little jobs and, distribute them out to the rest of the computers. Then each computer does the processing of those little tasks and, sends back the results of processing. The host computer takes all the results from the individual tasks and then it puts them all together again to generate the final result. In this project the main focus is the cluster computing system architecture.

4.1. Cluster computing systems

In this kind of architecture there are a collection of computers (also called nodes) linked together through a network [10], this enables computers to have a coordination of their activities and to share the resources of the system. Users typically perceive this architecture to be a single system.

This systems are very efficient due to its scalability and fault tolerance characteristic. They can easily allocate more users or respond faster to requests, just with adding more nodes to handle that extra load. They also avoid the problem of having a single point of failure, this is achieved by adding a good recovery and redundancy system.

Another characteristic of cluster computing is the transparency [11], users do not know when there is a failure or if there is a replication process or where are actually running different process.

Cluster mechanisms allow to have two or more process working together as a unique entity. In OpenDayLight is possible to have multiple instances of the controller working together as one entity.

Advantages:

- Scaling: if there is more than one controller running in a network, it will be able to work more and process a higher amount of data. It is also possible to have the data distributed across the cluster, this is known as sharding where is possible to have different shards on each controller of the cluster.
- High Availability: if one of the controllers goes down, it can be possible to continue running the network and being available.
- Data persistence: when a controller crash, it won't lose the data on it.

There are many uses and configurations, depending on the type of cluster that is needed. They can go from web services to scientific computations.

4.2. Load balancing

In this configuration all the nodes are sharing the workload of the cluster, this means that if there is a given service and it is receiving requests from many user then it can share all the load amount all the nodes just exactly how was described before. This configuration will provide a better performance of the network. If a node fails, the cluster will redirect all the load from that node to the other nodes. In that way the overall response time will be optimized.

4.3. High availability

They are also called HA cluster, the main idea is to create a redundant network to improve the availability of the cluster approach. This is perform to guarantee the service when a system component fails, giving multiple points of failures to the network. In this work this is the one that is going to be used.

5. Raft, consensus algorithm

Raft is a consensus algorithm for managing replicated logs [12]. Before to start to explain the algorithm it is necessary to define what consensus is?

Consensus is an algorithm which allows a set of nodes or servers to work together as a unique coherent system that is able to handle failures of some of its nodes. This can be done by replicating the state machine of the leader.

Each node has its own copy of the state machine but the system as a whole has the illusion that there is only one coherent state machine as is shown in Figure 5-1, even if some of the nodes are down. The distribution of the state machine can be used to solve different problems in large scale systems with single leader.

A typical consensus cluster can recover from a server failure autonomously, there are 2 cases for failures and they are the followings.

- Only the minority of servers fails. In this case the cluster can continue operating in the same way without having any problem
- The majority of the servers fails. In this case the cluster won't be available anymore until that a new majority of the servers runs again, but even with not availability the cluster will retain consistency of the information.

Replication is perform by using a replicate log, each node has its own log but they have to be identical to the ones of the other nodes, even in the same order.

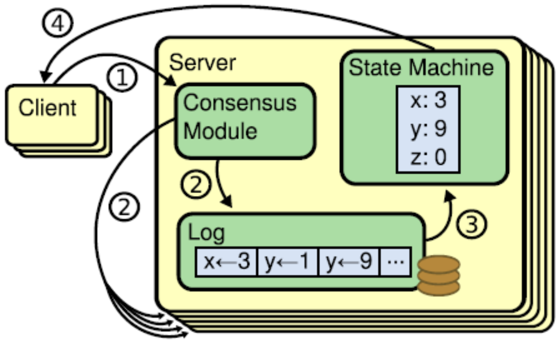


Figure 5-1 - Replicated state machine architecture. *[12]*

All the commands from clients are replicated in the other nodes, once those commands are replicated and processed for the nodes then the leader can send an answer to the client. For this reason nodes appear to be a single state machine.

All the consensus algorithms have the following characteristics:

- **Safety** is a big characteristic of consensus, this is because the system never gives an incorrect answer to clients, not even in condition of delay or packet loss.
- **Availability** is always guarantee when the majority of the nodes are operational active. This means that if there is a cluster of 7 nodes, the system can handle a failure of 3 nodes without losing availability.
- **Not time dependent,** ensuring the consistency of the logs in case of clock failure

The consensus is implemented first by electing a leader for the system, after the election the leader obtains all the responsibility for managing the replicated log. The procedure is the following, the leader receives log entries from the clients and, then it replicates them to the other nodes. When the majority receives and confirms the log entries from the leader, it informs to all the nodes to apply those log entries to their state machine or “commit the transaction”. If the leader fails for any reason, there has to be place for a new election.

It was used the consensus module to ensure the proper log replication. As was mentioned before, the system won’t make any progress as long as a majority of the servers are down.

Raft algorithm is splitted into three parts, Leader election, Log replication and Safety. It is going to be described every part of the algorithm in the same order.

In leader election the idea is to select one of the servers to act as a cluster leader and, if that server goes down there has to be place for a new election.

In log replication, the leader task is to take commands from clients, appends those commands to its log and, then replicate its logs to other nodes with the aim of make match its logs with the ones in other servers. This is done in order to overwrite inconsistencies.

In safety, the idea is to add restrictions to the leader election process, so only the server with the more updated log can become leader.

In a cluster architecture the typical number of nodes is 5, this gives the system the possibility to handle up to 2 failures. Raft implements 3 different states for the nodes of the cluster, they are leader, candidate and, follower as is shown in Figure 5-2. When a node is initiated it always starts at follower state that is a passive state, this means that it does not issue any request, it only responds to requests from leaders and candidates. The leader state handles all the requests from clients, if a client contacts to a follower it has to be redirected to the leader. The candidate state is the state in which a new leader is elected.

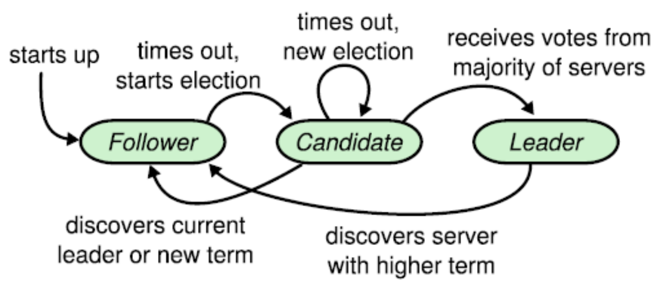

starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

Figure 5-2 - Server states. *[12]*

Another characteristic of Raft is that it divides time into **Terms** with arbitrary duration and, those terms are enumerated in a consecutive way as is shown in Figure 5-3.

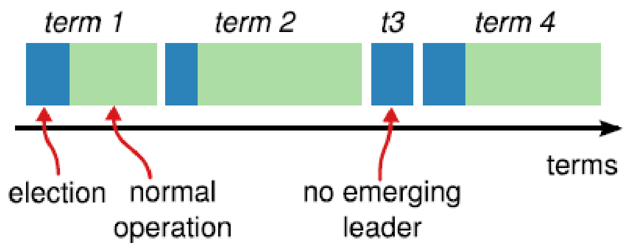


Figure 5-3 – Time Division *[12]*

A term always starts with an election process, where one or more nodes in the candidate state try to become leader. When one of the candidates wins the election and became leader, the term is conserve until the leader fails. There are also some situations, in which during a term there is not any leader election, this can be caused for a split vote’s situation. When this happens the term will ends up without any leader, then a new term starts in order to have a new election. This ensure that only a node can become leader within a given term.

The term value plays an important role in Raft, it is used for the nodes in order to detect obsolete information. The term is exchanged in any communication between nodes, the idea is that when a given node receives a larger term, it will update its term value to the one sent for the other node. When this happens the node immediately comes back to the follower state. The other important part of the term is when there is an election process, because when a node receives a vote request from a node with a smaller term, it will reject it.

There are two different messages in Raft, one is the "Request Vote" and the other is the "Append Entries", they both are RPC messages. The "Request Vote" is used for the candidate nodes in order to obtain votes from the other nodes and, the "Append Entries" is used for the leader in order to replicate log entries. When the "Append Entries" message is empty, it is called "heartbeat" message.

5.1. Leader election

When nodes are started up, they begin in the follower state. They remain in that state as long as they do not receive any message from a leader or candidate. If they don’t receive any message over a period of time called "Election Timeout" as is shown in Figure 5-4, then they will change their status to candidate, assuming that there is no current leader.

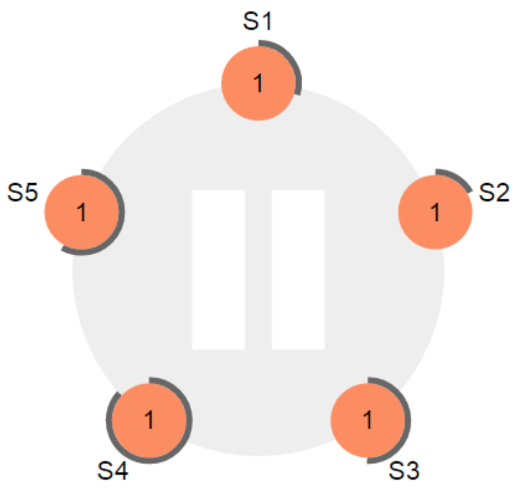


Figure 5-4 – Election Timeout *[13]*

As was mentioned before if a node in follower state wants to start a new election, after timing out it has to increment the term and change its state to candidate. When election process starts, the candidate node always votes for itself and sends a "Request Vote" message to try to obtain votes from the other members as shown in Figure 5-5.

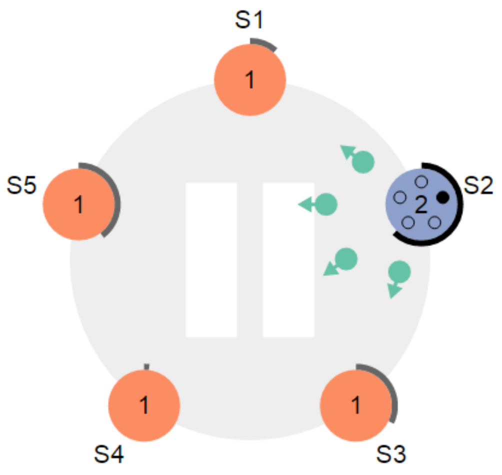


Figure 5-5 – Vote Requests *[13]*

The candidate node remains in that state until:

1 It gets the majority of the votes and becomes leader of the cluster.

When a candidate wins an election is because it got the majority of the votes, this is done in a first come first served basic, as is describe in [12]. This ensure that only one node can become leader in a given term. When it becomes leader it starts to send "heartbeats" to the other nodes in order to inform that there is a leader and prevent new elections.

2 Another node becomes leader by sending a “heartbeat” message.

If a candidate node receives a "Heartbeat" or an "Append Entries" message while it is waiting for the votes, it will come back to follower state because there is still a leader in the cluster.

3 No one gets the leadership and a new term has to start.

In the case of having multiples nodes changing their state to candidate at the same time, it could ends up in a split vote situation. When this happens the candidates will start a new election in the next term.

Figure 5-6 shows how nodes vote for a candidate.

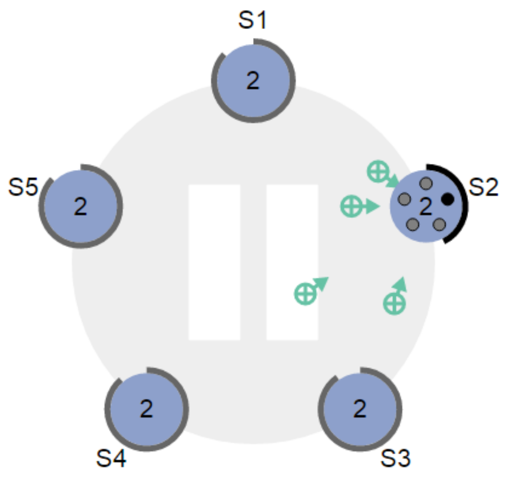


Figure 5-6 - Granted Votes *[13]*

In the third case above mentioned it may be a situation with split votes indefinitely, that's why Raft uses extra measures to prevent this. It uses randomized election timeouts in order to solve this problem. They are typically between 150-300 ms, in this way only one server will timeout in a given term. It will also sends "Heartbeats" and receives confirmation before another node timeout.

5.2. Log replication

After a leader is elected, it can start to serve clients. Those clients make requests which contains commands to be append to the leader log, in parallel it is also sent

"Append Entries" to the other nodes in order to perform the replication. When that entry is safety replicated to the majority, the leader can finally apply it to its state machine. This process is also called “commitment process", which means that a given command has been replicated and applied to the state machine and now is safe. The entry is durable and will never be overwrite.

The way Raft organizes logs is the following, each log entry has a command along with a term number which says when the entry was received by the leader. The term number is really important because with it the system detects inconsistencies in logs. Each log entry also has an integer index in order to identify its position in the log as is shown in Figure 5-7.

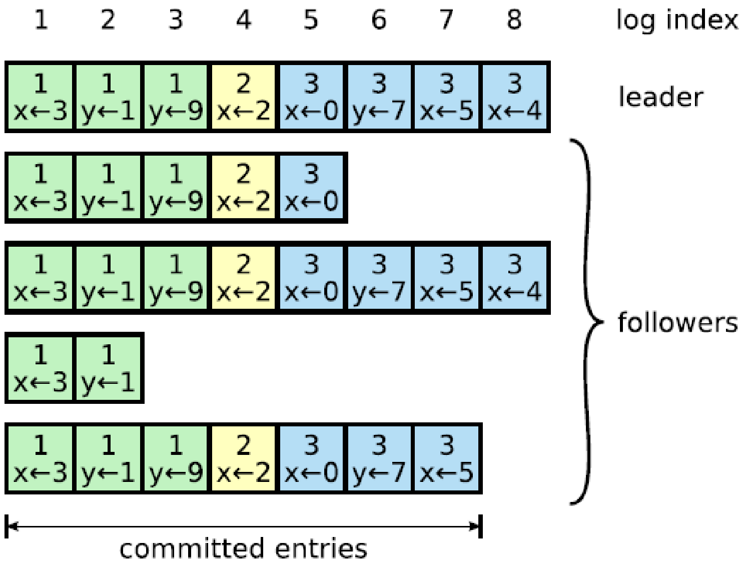


Figure 5-7 – Log Entries *[12]*

Another properties of Raft are: log entries never change their position in the log in order to have consistency and it also perform consistency check. This is done by using "Append Entries" messages. Within the message is include the term and log index of the entry preceding the new entry. This information is used for the follower node. If the follower does not find any entry with that term and log index, then it will drop the new entry. But if the follower finds the entry with that term and log index, that means that the leader log and the follower log are exactly the same. It will return a success "Append Entries" message to the leader.

When the system operates normally, the consistency check always returns a success operation which means that the log of the leader and followers stays consistent. In case of inconsistency in the operation of the system, it’s possible to be in a situation of leader or followers crash, which leads to logs inconsistent.

Log inconsistency means that follower’s logs may be different to the logs in the leader, it can be in different ways. Having extra entries or having missing entries, those inconsistencies are solved by overwriting conflicting entries with entries from the leader, this is a safe method but it has to be done with some restrictions that will be explained in the safety part.

The procedure is the following: the leader has to figure out which was the last entry where there is a match between its logs and the follower logs in order to delete the other entries after that point in the follower node, then sends all the entries after that point from the leader log.

Operating in normal conditions, a new entry is replicated within a single round of messages to the majority of the nodes.

5.3. Safety

After describing how Raft performs leader election and how it replicates logs, it’s also necessary to talk about some mechanisms to ensure that all the state machines to be the same in all the nodes. For example, when a leader is committing log entries while one of the nodes is unavailable and after sometime, that node is elected leader. It starts overwriting the entries committed for the previous leader with new entries. In the above case it may result in a loose of consistency and it is necessary to apply some restrictions in order to ensure that the leader for any given term contains all the entries committed in previous terms.

5.3.1. Election Restriction

in Raft all the committed entries from previous terms have to be present in new leaders at the moment of its election, this means that there is no need to transfer any entry to the leader, this follows the rule that log entries only flow from leader to followers and not vice versa. A leader never overwrites log entries that are already in its log.

They way in which Raft prevents candidates without previous committed entries to become leader is during the voting process. When a candidate node makes contact with other nodes in the system for a Request Vote, the message includes information about the candidate log. This is used for the nodes in order to determine which log entry is more up to date, the one of the leader or its own. If the follower has a log more up to date with respect to the one in the leader, it will deny the vote and, if it has the log less up to date then it will confirm the vote.

With this procedure is possible to ensure that the leader elected has all the log entries committed in previous terms.

5.3.2. Committing Entries from Previous Terms

An entry is said to be committed once is replicated to the majority. When a node is doing its replication duty and it crashes before it can commit a given entry. That entry won't be safe of being overwritten for future leaders. Future leaders will try to finish replicating the entry, but unfortunately a new leader is not able to know if an entry from previous terms is committed.

In order to solve this problem, Raft never commits log entries from previous terms. Only the entries replicated from the current leader can be committed, this is a way to ensure that prior entries are being committed too.

5.4. Summary (Raft)

In election when a shard starts up, it starts as a follower because it does not know about anything else in the system. After that it waits for some time in terms of heartbeats, if it doesn't receive any heartbeat from the leader in that time period then it becomes a candidate and starts to send requests for votes. After that it waits for the votes and of course it votes for itself and, if it gets the majority of the votes it'll become the leader.

Once it becomes the leader then it has the authority to replicate data to the other nodes, it happens by sending "Append Entries" messages. Those messages can also act as a "Heartbeat" when they don't have any payload as a way to replicate data, that's how replication works.

The consensus works in the following way, if there is for example a leader and two followers. It is necessary to be able to replicate to at least one follower and, see the confirmation from it saying that the data has been stored. Then the leader can finally commit it and put it into the data tree, if the leader doesn't get any response it cannot put it into the data tree and that information stays in the journal.

5.4.1. Journal Replication

When there is a transaction process on the cluster and, it gets replicated, the leader node needs to know if the majority of the nodes got the transaction before that the leader can confirm it and put it into the data tree as is shown in Figure 5-8. The cluster cannot tolerate commit transactions while the majority of nodes are down, in this case it will have all the transactions coming and they will be stored into the journal but they will never be apply to the data tree.

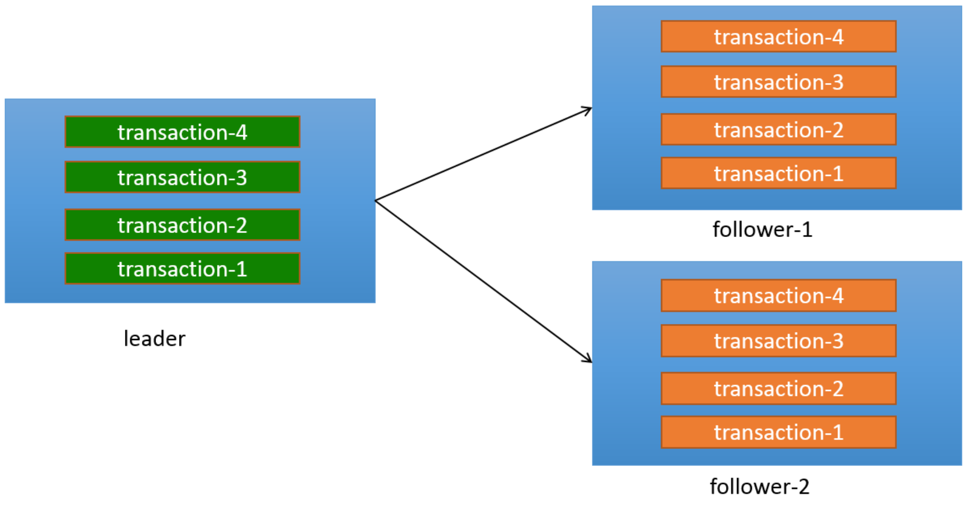


Figure 5-8 – Journal Replication *[14]*

5.4.2. Snapshot Replication

Typically when a cluster brings up a node, it's not very efficient to send one by one the entries to that node for complete the replication because it will take too much time. So essentially when a node restarts instead of sending "Append Entries" it just sends the snapshot as is shown in Figure 5-9. So the whole data tree is sent, in addition to this it also breaks up the data tree in smaller chunks to perform the replication because normally this data tree can be really large. The size of those chunks are typically fixed to 2 Mb.

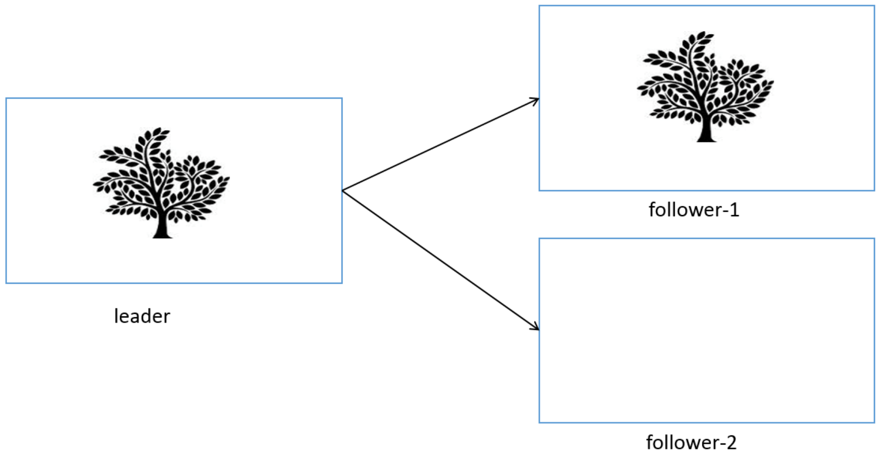


Figure 5-9 – Snapshot Replication *[14]*

5.4.3. Durability/Recovery

Durability is useful in the recovery process, in Figure 5-10 there are two components: the first one is the data tree that is stored in memory but there is also the journal which is persisted and the reason for this is that when there is a restart, a node has to recover from persistence. For example, if there is a configuration data and a bunch of flows is added into the configuration then when there is a controller restart, it is desired to see all those flows in there because otherwise it won't be able to reconfigure all the switches in the same way. So the journal is essentially all the modification that were ever made and stored in the journal one by one and, the snapshot is used because it's so important to recover faster. For example, when there are thousands of flows in the journal, it's not so good to wait a long time for each flow to be read from disk and then be added it into the data tree. It's just better to put all the data tree into a snapshot that is a disk file and, it will read all at once and from it will be construct the data tree.

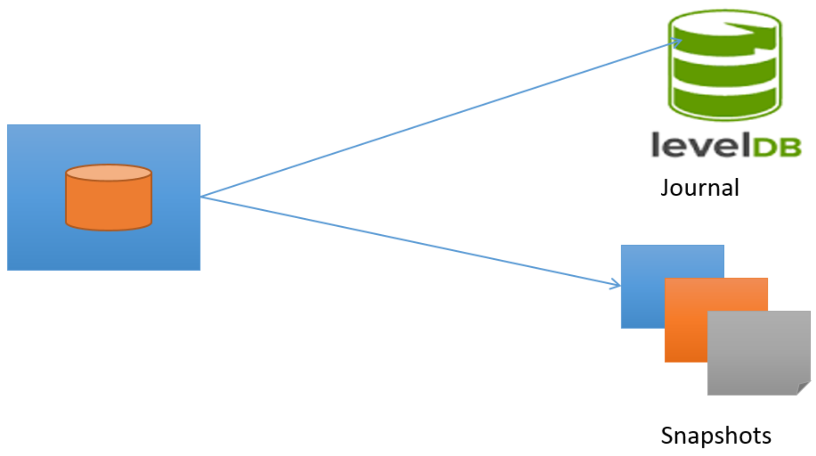


Figure 5-10 - Durability/Recovery *[14]*

6. Akka

According to its specification, Akka Cluster provides a fault-tolerant decentralized peer-to-peer based cluster membership service with no single point of failure or single point of bottleneck. It does this using gossip protocols and an automatic failure detector [15].

It’s important to explain some definitions before get more in deep with Akka [16].

- Node: A logical member of a cluster.
- Cluster: A set of member nodes joined together through a service.
- Leader: A single node in the cluster which acts as a Master. When a node is the leader, it has full access to the switch.

6.1. Membership

A cluster is compose for a set of logical nodes, each node is identified for its hostname , port, and an identifier number UID that is given by the system with the aim of differentiate all the members and provide a better control in join and death process. The membership process is initiated by sending a "join message" to one of the seed nodes in the system, this communication between members is performed by using the **Gossip Protocol**. The current state of the cluster is gossiped in a random way to the members in the cluster, with some priority to the members that have not seen the updated version of the state.

6.2. Failure detector

The failure detector is in charge of detecting unreachable nodes within the cluster. The way in which it acts is the following: the idea is to keep the history of failures statistics. This is calculated from heartbeats received from other nodes, having that in mind was created a threshold to count how many failures are necessary to declare a node as unreachable, the variable is called the phi accrual failure detector and it can be configurable by the user. It is an important thing for the system because with a high threshold it may leads to a situation with few mistakes but it will need more time to detect real crashes, instead with a low threshold it may generate more mistakes but it will give a fastest detection. The default value for this variable is 8 failures and is appropriate for most situations.

In cluster architectures a node is typically monitored by a few of the other nodes, it depends in the cluster size but normally the maximum number is 5 nodes that monitors a single node. So when a given node detect that another node is unreachable, it will send that information to the rest of the nodes by using the gossip protocol, that's why only one node needs to mark a node as unreachable to make the rest of nodes to mark that node as unreachable too.

The other function of the failure detector is to detect when an unreachable node becomes reachable again, this is again done with a gossip round.

6.3. Seed nodes

Seed nodes are in charge of being the contact points for all the new nodes that are joining the cluster, its address and hostname have to be stored in all the nodes that are willing to join the cluster. When a new node try to join a cluster the first thing to do is to contact seed nodes, after that it has to send a join command to the seed node which answered first. It is possible to configure many seed nodes in the cluster, but with only one seed node the cluster can works pretty well.

Seed nodes do not have any influence in the cluster performance, they only act as a contact point for new nodes.

6.4. Membership lifecycle

In Figure 6-1 is shown the membership lifecycle. When a node starts, it always starts in the joining state. This happens until that all the nodes get to know that there is a new node, this is done through gossip convergence. Once the convergence is achieve the leader changes the status of the new node to up.

In the case which a node is leaving the cluster in a correct way, the leader changes the status of that node to a leaving state and then when the system achieves convergence, it will move the node to an existing state and then will mark it as removed.

In the case of an unreachable node, the system won’t be able to achieve convergence and therefore won’t be able to move forward. In this case the leader waits for that node to become reachable again or definitely marks it as down.

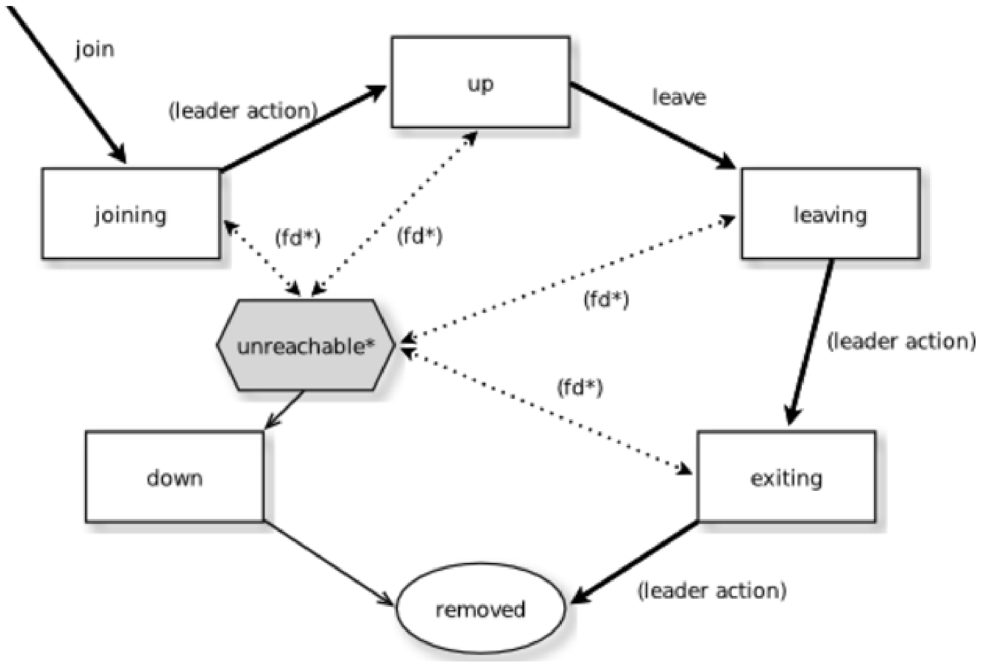


Figure 6-1 – Membership Lifecycle *[16]*

Member states are the following: joining, up, leaving, exiting, down and, removed. There are also some actions for the leader and users.

- User actions: join, leave and, down.
- Leader actions: joining and exiting.

6.5. Joining to seed nodes

In the cluster architecture it is necessary to configure initial contact points, this is done with the aim of make an automatically joining for all the nodes that will try to join the cluster. Those contact point are also called seed nodes. The idea is that when a node starts. It has to go and read the seed node list in the configuration files, this is done in order to obtain all the addresses of seed nodes and try to contact them. When a "join node" gets an answer from any of the seed nodes, it has to send a join command to the one that replied first, in this way is initiated the joining process. If no one answered, the node has to continue contacting seed nodes until it gets an answer or until it shutdown.

Seed nodes can be started in any order, but the only condition is that the first node in the seed node list has to be the first one to start in the cluster. Otherwise it won’t be able to initialize other seed nodes. The reason for this is to avoid the creation of separated islands in the cluster. There is not any restriction on how many seed nodes have to be started, but it’s needed at least 2 seed nodes to start the cluster. Once there are more than 2 seed node running, it will possible to shut down the first node in the seed node list

When the cluster is formed, a new node can try to join the cluster to any member node. Even if that node is not a seed node.

The first node in the seed node list will join itself in case in which it cannot contact any other seed node

6.6. Leaving

In Akka there are only two ways to remove members from the system. The first is stopping the node and then wait the other nodes to detect the node as unreachable, after that the leader will mark it as removed. The second one is informing the system that a given node has to leave.

6.7. Node roles

In distributed systems all the nodes have different roles, this means that the workload can be distributed in any way to each member of the system. For example, one node can be in charge of the Data Store, another of the architecture and another of the inventory. But there is also the possibility to give the same role to all the members by having redundancy, this can be done by replicating all the duties among the members in order to gain availability in the system. The roles are defined in the configuration files.

6.8. Persistence module

The persistence module of Akka allows nodes to persist their internal states, this is done in order to allow recovery when a node is started or restarted after a crash. In reality only changes to the internal state are saved, so the changes are persisted but never its current state (except when there is also the snapshot implemented). Those changes are saved in memory but they are never mutated, this make a highest transaction rate and more efficient replication. All the persisted changes are saved into a journal in order to be replayed later with the aim of recover the internal state from all the messages.

When a node needs to recover, it has to replay all the stored changes in order to rebuild the internal state. It can start to rebuild the internal state from zero or start from a snapshot which will make the process fastest, reducing the recovery time.

A node is automatically recovered by default on start or restart, this is done replying all the messages in the journal. If during the recovery phase new messages are sent to the node, they won’t interfere with replayed process. They are stored in cache and when the recovery phase ends they will be processed.

6.9. Snapshots

With the use of snapshots the system can reduce dramatically recovery time. The system saved snapshots of the internal state in order to use it later during recovery. This snapshot is offered when a node is about to start or restart with the aim of initialize the internal state. If there are several snapshots in the system, it will take the youngest version.

7. Gossip protocol

Akka uses a version of gossip call push-pull [16], this is with the aim of reducing the information sent in the cluster. This means that it only sends current versions but not actual information. In the answer to a gossip message a node sends another value that represents if that node has an updated version or an outdate version. This is done with help of the vector clock for the versioning, making possible only to pull the information as needed.

Each node has the bucket or version information for all the other nodes as is shown in Figure 7-1. For example in a cluster with 3 members, member one has the information of member two and three. In this way when a member does a gossip with another node, it can say if it has an older or a newer version with respect to the other nodes. The gossip mechanism works in the following way, every second all the members send status messages to each other, saying which version of the bucket they know. Then with based on that information, nodes can decide to send back their status if their version are lower or otherwise they send an update. Every time that something changes in the bucket is changed also the version of the bucket.

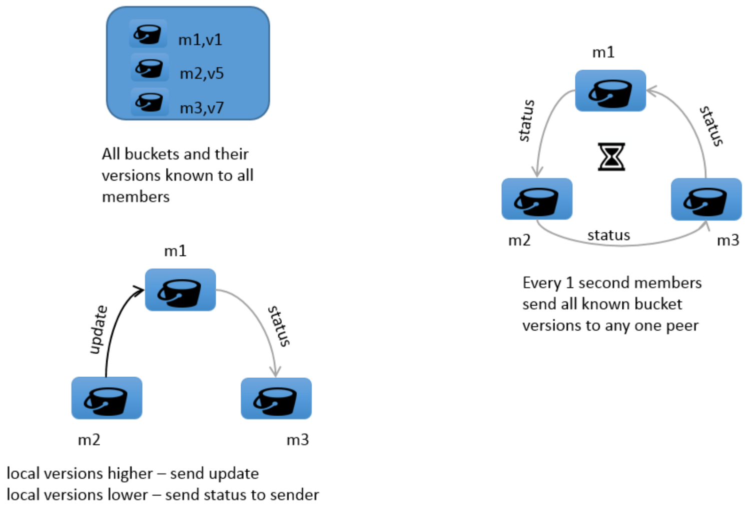

m1,v1
m2,v5
m3,v7
All buckets and their versions known to all members
m1
m2
m3
update
status
local versions higher – send update
local versions lower – send status to sender
m1
m2
m3
status
status
status
Every 1 second members send all known bucket versions to any one peer

Figure 7-1 – Gossip Protocol *[14]*

Messages are exchanged normally every 1 second and the decision of where to send next the gossip message is random, but there is still some priority to those nodes who have not seen the last version.

Another concept that needs to be explained is the concept of gossip convergence. It is possible to talk about convergence when a member of the cluster can surely prove that the cluster state that he is observing has been observed for all the other members of the cluster. This information can be pass from node to node, each time a node receives a gossip message it reads which members have already seen the cluster state and, it also marks it before to send a new gossip message. In order to have gossip convergence it is necessary to have all the nodes available, it is impossible to achieve convergence while there is a node as unreachable.

When the cluster is in a convergence state, the system only sends small gossip messages containing only the gossip version. But when there is a change in the cluster and there is not convergence then the system goes back.

Gossip protocol also makes use of an algorithm for the data structure, this is called Vector clock and it does a partial ordering of events and detection of violations in distributed systems. Gossip uses the vector clock in order to note differences in the cluster state when there is a gossiped exchange. A vector clock is a couple of (node, counter) pair, so every time the cluster state change, then the vector clock also has to update.

The vector clock is used to determine the following:

- If the sender has a newer version, in this case it sends back a message in order to request the new version.
- If the sender has an outdated version, in this case the recipient sends back its gossip state.
- If there Is a conflicting gossip versions, in this case those versions are merged and sent back.

8. Clustering in OpenDayLight

Starting with the architecture

Essentially there are 2 subsystems implemented, in the cluster implementation:

- Data Store

The idea behind the Data Store implementation is that it is allows for high availability and scalability, where all the members are talking to each other distributing the data as shown in Figure 8-1.

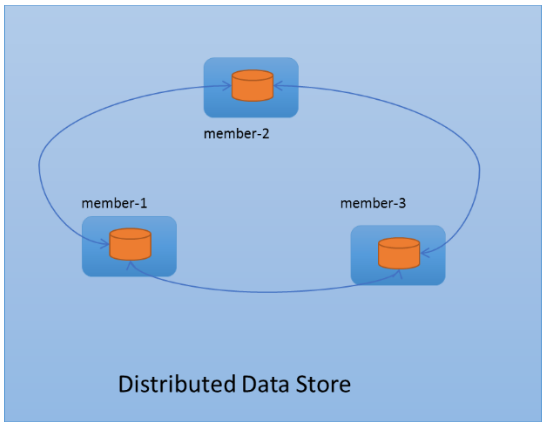


Figure 8-1 – Data Store *[14]*

- Rpc

If there is a router RPC and it is trying to registerer in a node, it is invoked that RPC either from RestConf or from any node in the cluster regardless of where is been invoked from.

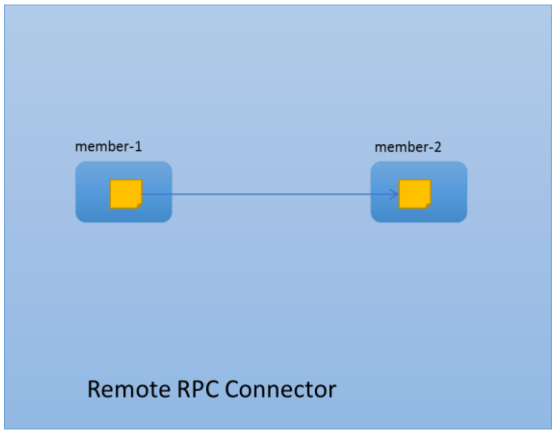


Figure 8-2 – RPC *[14]*

8.1. How is this build on?

At the high level architecture, both the Distributed Data Store and the RPC connector have more or less the same foundation, everything was built based on Akka actors as shown in Figure 8-3. That is a way to ensure that these components can reside on any node in the cluster, the actors can be run anywhere and it is possible to invoke these actors or send messages to these actors from any node in the cluster without build the intelligence of where exactly the actor reside.

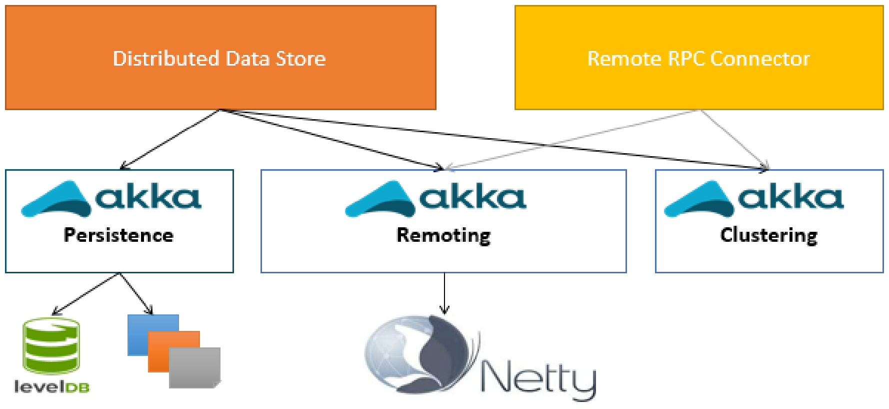


Figure 8-3 – Akka Actors *[14]*

The Akka persistence module, gives the ability to store data and in case of restart, the node gets back the data and it reconstructs the state of the tree, it has 2 different things:

- The Journal, is essentially a file with all the modification that you ever made in the data tree.
- Snapshots, is where the whole state of the tree is stored.

The Akka remoting module, is a module in which an actor system in one node communicates to another actor system in another node, that's the main idea.

The Akka clustering module, is used for the discovery of nodes. For example, if there are 2 nodes and it is wanted to know where the other node is or which is the ip address or where is hosted. Akka clustering will give that information. It also gives information about the status of members, for example, if "is the member alive, dead, reachable or not reachable".

8.2. Data synchronization

In the **data store** there are trees and the objective is to have all the trees synchronized as is shown in Figure 8-4. There are different data trees like inventory, topology, Toaster. These trees are allocated in a big tree, which is synchronized for high availability. For this is used an algorithm called "RAFT the consensus algorithm" to make sure that all these trees look the same on each node.

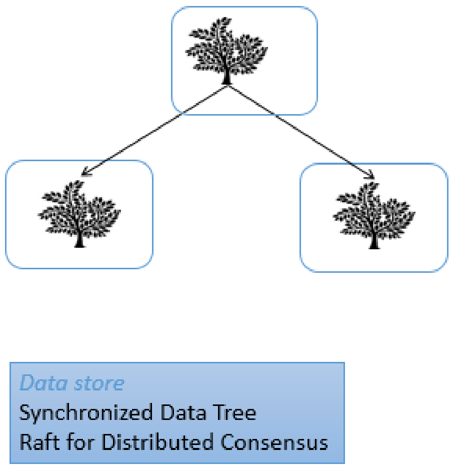


Figure 8-4 – Synchronized Data Tree *[14]*

For **RPC**, there is a synchronization of the RPC Registry for each node that gets registered, for example, an open flow switch who wants to add a RPC flow on node 1, it is important to know exactly how to invoke the added flow on that switch. That information goes into the registry and it gets also replicated as shown in Figure 8-5.

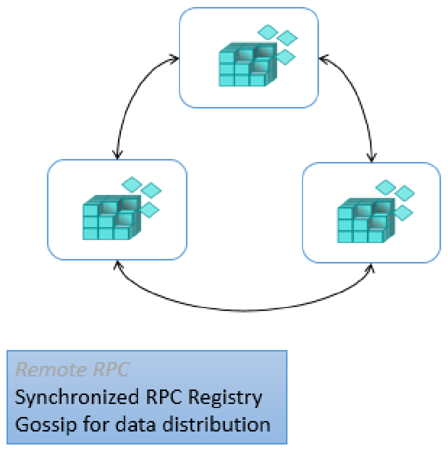


Figure 8-5 - Synchronized RPC Registry *[14]*

8.3. Communication

The way in which the "distributed data store" communicates with the data tree, is putting an actor around the data tree. So when is desired to have communication with the data tree, it is just necessary to send a message to the actor and wait for that message to be processed, leading to a modification of the data tree.

8.4. Data Distribution

Once it is possible to access data remotely, the next thing to do is to not have all the data in the same place. In the case of a big data tree, it is broken up into smaller trees and distributed across the cluster as is shown in Figure 8-6.

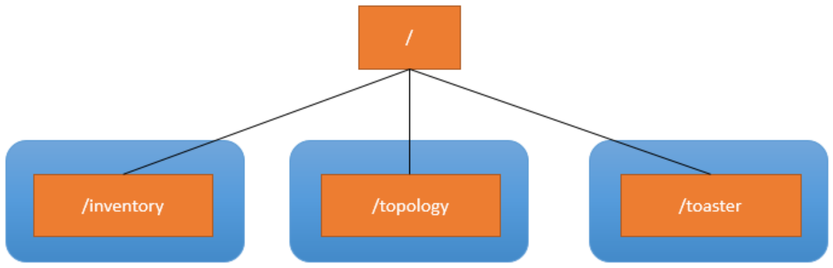


Figure 8-6 – Sharding *[14]*

8.5. High Availability (HA)

With the distributed data store architecture it is possible to distribute all the data, for example, inventory can go on one node and topology in another node. But what would happen if inventory is in one node and that node fail?. Then the system would lose access to the inventory data.

The solution for this problem is to distribute also the data across the cluster in a replicated way as is shown in Figure 8-7. For example, member 1 is the leader of a given shard and, member 2 and 3 act as followers, these followers have exactly the same data. So that if member 1 goes down one of the other two nodes will take the role of leader in order to guarantee high availability. The way it works is again governed by RAFT algorithm.

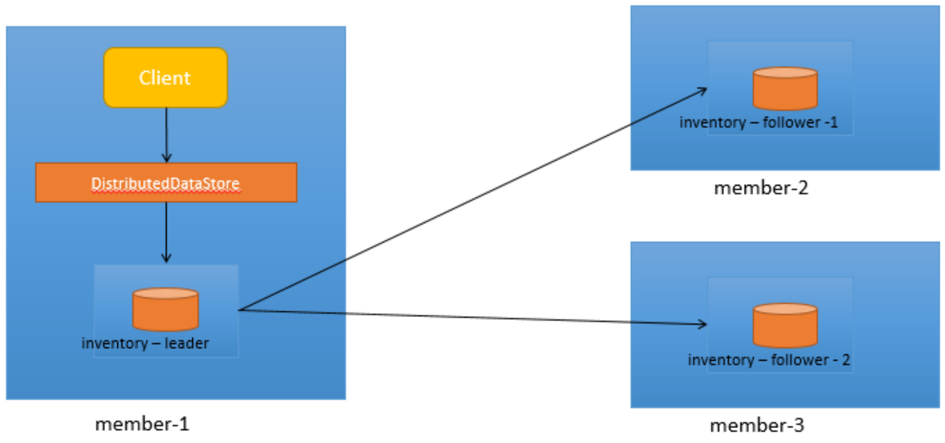


Figure 8-7 – Distribute Data Store *[14]*

8.6. Data Store Flows

There are 2 modules in the data store, one is de Config data store and the other is the operational data store.

8.7. Startup

Once the cluster starts up, it comes created an instant method of the distribute data store and then it has to wait until it gets ready. The problem here is that when there is a distributed data store, it is difficult to know when it is ready for use?. For example, with the "In Memory" data is quite clear, it creates the "In Memory" data store and, it is immediately available for work, it can start creating transactions and so on. Instead In the "Distribute" data store, that’s not possible because if there is one instance of the controller started and the other instances have not started then consensus will not be there. For example, as who is the leader.

So after the system has created the distribute data store there is a waiting time until it gets ready, normally for 90 seconds trying to find the leader. It’s enough time to start another node and for the creation of its shards, this in order to select the leader. If that happens within 90 second then it will move forward, otherwise it will block for 90 seconds. Once it is create the distribute data store, it creates two classes: one is the ACTOR CONTEXT that allows to communicate with actors, this is necessary because distribute data store is not an actor and, the other one is SHARD MANAGER which is the parent of all the shards. Shards are created in base on a configuration file called module-shards.conf, so for all those shards is necessary to find the leader within the 90 seconds as is mentioned above. As a result if a transaction is created before a leader is found, that transaction will fail.

When shards are created, the first thing they do is to read and recover the information from disk. It can happen either from the journal or the snapshot, it goes and reconstruct the data tree and then it sets its behavior to follower, after that it says "I am ready for communications" to continue with the process of elections in order to choose the leader. Once the leader is found, the countdown and the wait until it gets ready happens. It can move forward, this procedure needs to happen for both ConfigDataStore and OperationalDataStore.

9. Set Up and Testing of the Cluster

In this chapter is going to be shown which is the configuration needed to deploy a cluster architecture with the OpenDayLight controller.

The idea is to have multiple instances of the controller working together as if they were a single entity, in order to achieve a better scaling, persistence and, high availability in the system. In this case it will be a 3 node cluster as shown in Figure 9-1, in which the base distribution is Helium-SR4

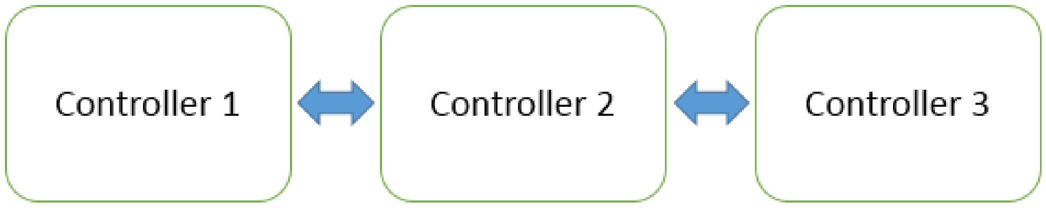


Figure 9-1 - 3 Nodes Cluster

Before getting more in deep with the cluster deployment, it is necessary to make some considerations.

When it is wanted to build a clustered system, it is really important to have in mind that it will be needed an odd number of controllers. This is because OpenDayLight is using the Raft algorithm and it always needs to have a majority of the members available, in order to maintain the system in a high availability fashion. This means that the minimum cluster size to maintain the HA feature is 3 nodes. If it is built a cluster of less than 2 nodes, it can be possible to make some functional test but it will never be in high availability.

Another thing that is needed to have in mind is what is the role that every node is going to play, this is important because it has to be configured previously to the running. In this case as the main objective is to create a HA cluster all the members are going to play the same role.

The last consideration is to know which is the data needed for the system, this data is allocated among different shards. By default OpenDayLight brings some shards already created and they are: Inventory, Topology, Toaster and a default one for the other kind of information. In this thesis are going to be used only these shards because for the thesis purposes is not necessary to create a new one.

The first thing to do is to create a Virtual Machine (VM) where to host the controller, in this case it is used a version of Ubuntu 14.04 [17] and VirtualBox 5.0.6 [18].

Now that there is a machine where to host the controller, the next step is to get the OpenDayLight controller for this parte there are two options: the first is to download a version that is already modified with all the features needed, this version can be found in some OpenDayLight repositories. The second option is to download the original version and add manually all the features needed for the cluster deployment, this version can be found from the OpenDayLight webpage [8].

For this case was used an original version downloaded directly from the OpenDayLight webpage. Here is not going to be explained how to install and configure the VM [19]. The features that are going to be needed are odl-restconf, odl-l2switch-switch, odl-mdsal-clustering, odl-openflowplugin-flow-services.

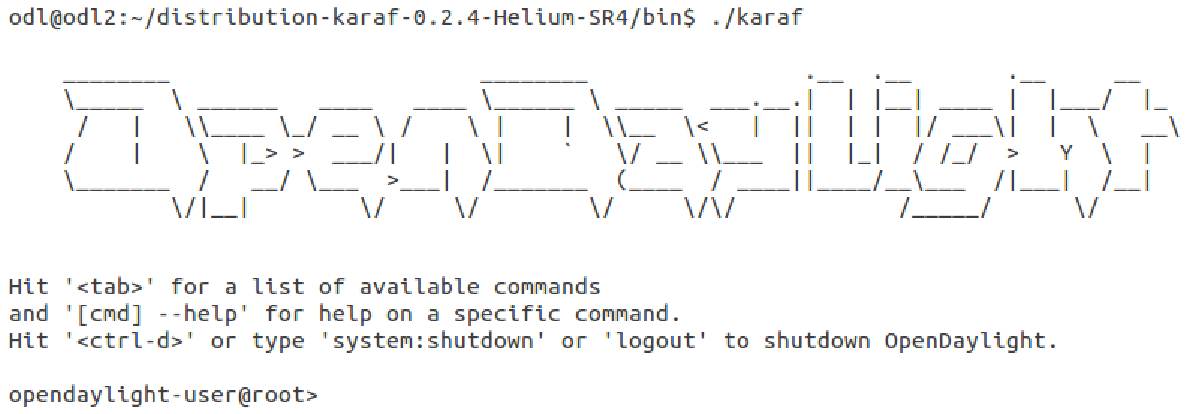
In the cluster architecture it is going to be needed at least 3 VM, one for each controller. Then is possible to run the controller in each node, this can be done by entering in the distribution folder and, looking for the <Karaf-distribution-location>/bin directory.

```
odl@odl2:~/distribution-karaf-0.2.4-Helium-SR4/bin$
```

In order to run the controller it is necessary to execute the file Karaf like this ./Karaf.

```
odl@odl2:~/distribution-karaf-0.2.4-Helium-SR4/bin$ ./Karaf
```

This will run the controller and will appear a command window as is shown in Figure 9-2.



```
odl@odl2:~/distribution-karaf-0.2.4-Helium-SR4/bin$ ./karaf

    ________                       ________                .__  .__       .__     __
    \_____  \ ______   ____   ____ \______ \ _____  ___.__.|  | |__| ____ |  |___/  |_
     /   |   \\____ \_/ __ \ /    \ |    |  \\__  \<   |  ||  | |  |/ ___\|  |  \   __\
    /    |    \  |_> >  ___/|   |  \|    `   \/ __ \\___  ||  |_|  / /_/  >   Y  \  |
    \_______  /   __/ \___  >___|  /_______  (____  / ____||____/__\___  /|___|  /__|
            \/|__|        \/     \/        \/     \/\/            /_____/      \/

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type 'system:shutdown' or 'logout' to shutdown OpenDaylight.

opendaylight-user@root>
```

Figure 9-2 – Command Window

In this window is possible to perform some actions, like install features, consult which features are already installed and also to see some of the internal process of the controller. For example, when a switch is requesting a connection to the controller.

Now that the controller is in operation it is possible to install the features, the way it is done is the following: In the command window is typed feature: install and the name of the feature desired.

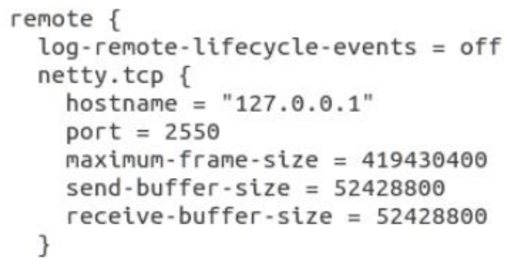
In this case the most important feature is the **odl-mdsal-clustering**, this will install the Akka toolkit and more characteristics in the controller and it will be the one making possible the cluster functionality. This feature will also create some initial configuration and, some files that are going to make possible the manual configurations. Those files are stored in the folder "configuration/initial" and they are named akka.conf and module-shards.conf*.*

It's also necessary to install the **odl-openflowplugin-flow-services**, this feature is the in charge of the communication with Open Flow equipment and in general of the communication between controllers.

Once the features mentioned above are installed, it is possible to start with the manual configurations of the nodes.

First, it is needed to go to the akka.conf file and make some modifications, the changes needed to do is to set the ip address and port in which the node will be listening, configure all the seed nodes and define the role of that node.

The ip address and port that are in the initial configuration are the following.



```
remote {
  log-remote-lifecycle-events = off
  netty.tcp {
    hostname = "127.0.0.1"
    port = 2550
    maximum-frame-size = 419430400
    send-buffer-size = 52428800
    receive-buffer-size = 52428800
  }
```

Then it is necessary to modify only the hostname address for the current ip address of the node, the port is the same. This configuration tells the system that it will be listening in the ip address 192.168.56.101 and port 2500, this is important because in this address and port will be received all the requests from joining nodes.

```
remote {
  log-remote-lifecycle-events = off
  netty.tcp {
    hostname = "192.168.56.101"
    port = 2550
    maximum-frame-size = 419430400
    send-buffer-size = 52428800
    receive-buffer-size = 52428800
  }
```

For the seed node list, initially there is the following.

```
cluster {
  seed-nodes = ["akka.tcp://opendaylight-cluster-data@127.0.0.1:2550"]
```

In this part it is also necessary to modify the address of the seed node because initially it is configured to contact itself to the localhost address. It is possible to configure more than 1 seed node in the system, there is not limitation on how many seed nodes have to be in the cluster.

```
cluster {
  seed-nodes = ["akka.tcp://opendaylight-cluster-data@192.168.56.101:2550","akka.tcp://opendaylight-cluster-data@192.168.56.101:2550"]
```

For the role of each node, initially it comes by defect as the following.

```
roles = [
  "member-1"
```

In this part it has to be modified only when the configured node is not the first node, but if it is the second it has to be changed to member-1 or member-2.

These are all the changes that are needed in the akka.conf file.

Then is also needed to modify the second file module-shards.conf. In this file it is need to specify if any shard will be replicated in other nodes. In this will be shown the case of the inventory shard, the initial file is like the following.

```
name = "inventory"
shards = [
    {
        name="inventory"
        replicas = [
            "member-1"
        ]
    }
```

It is possible to run the cluster without change any of this file, but if is looked to implement the case of High availability, it is need to make replicas of all the shards in all the nodes that are going to be part of the cluster like the following.

```
name = "inventory"
shards = [
    {
        name="inventory"
        replicas = [
            "member-1",
            "member-2",
            "member-3"
        ]
    }
```

Once is done the previous configuration in all the nodes, then is possible to restart again all the controllers and start with the clustering services.

9.1. Testing

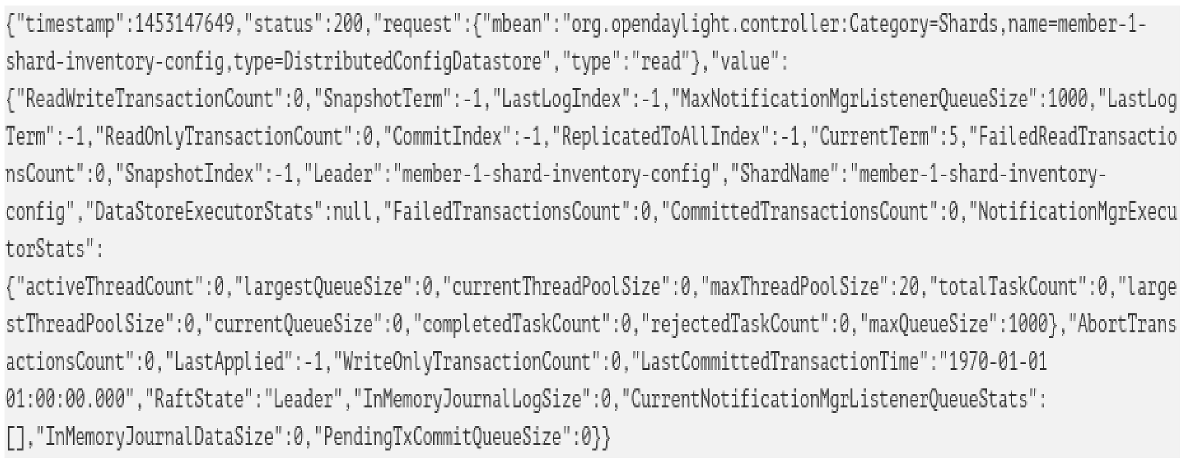
After the cluster configuration is necessary to implement some mechanism that allows to validate that the setup is right. Validate means to prove that the cluster is running properly, like validate that there is a leader for each shard and that the system is making the right operations like shard replication and committing actions.

For this purpose is going to be used "Postman" that is an application for making HTTP requests, with this app is possible to ask the controller for information about a specific shard. The command to do that is the following.

GET

http://192.168.56.101:8181/jolokia/read/org.opendaylight.controller:Category=Shards,name=member-1-shard-inventory-config,type=DistributedConfigDatastore

This request gives back information about the state of the cluster as is shown in Figure 9-3.



```
{"timestamp":1453147649,"status":200,"request":{"mbean":"org.opendaylight.controller:Category=Shards,name=member-1-
shard-inventory-config,type=DistributedConfigDatastore","type":"read"},"value":
{"ReadWriteTransactionCount":0,"SnapshotTerm":-1,"LastLogIndex":-1,"MaxNotificationMgrListenerQueueSize":1000,"LastLog
Term":-1,"ReadOnlyTransactionCount":0,"CommitIndex":-1,"ReplicatedToAllIndex":-1,"CurrentTerm":5,"FailedReadTransactio
nsCount":0,"SnapshotIndex":-1,"Leader":"member-1-shard-inventory-config","ShardName":"member-1-shard-inventory-
config","DataStoreExecutorStats":null,"FailedTransactionsCount":0,"CommittedTransactionsCount":0,"NotificationMgrExecu
torStats":
{"activeThreadCount":0,"largestQueueSize":0,"currentThreadPoolSize":0,"maxThreadPoolSize":20,"totalTaskCount":0,"large
stThreadPoolSize":0,"currentQueueSize":0,"completedTaskCount":0,"rejectedTaskCount":0,"maxQueueSize":1000},"AbortTrans
actionsCount":0,"LastApplied":-1,"WriteOnlyTransactionCount":0,"LastCommittedTransactionTime":"1970-01-01
01:00:00.000","RaftState":"Leader","InMemoryJournalLogSize":0,"CurrentNotificationMgrListenerQueueStats":
[],"InMemoryJournalDataSize":0,"PendingTxCommitQueueSize":0}}
```

Figure 9-3 – Testing Response

In this answer is possible to obtain valuable information as the last log index, current term, failed transactions, committed transactions and the current leader.

10. Log Analysis

In this part is going to be analyzed which are the actions that are performed for the controller in every node. When a controller starts its operation, it creates a text file which specifies all the steps that are made, this file is located in the folder "data" and the name is Log.txt. This means that this file contains all the information about actions, notifications and modifications made in the controller. Logs can help to understand what is really happening in the system.

In order to focus in the main goal that is the clustering part, is going to be analyzed only the information related to that feature.

Node 1

Let's start with the first node in the seed node list (Node 1). Once the controller starts and execute all the default features, it has to initialize itself as a cluster node. It's an automatically process as is shown in the following.

```
2016-01-19 18:04:11,359 | INFO  | ult-dispatcher-2 | Remoting | 266 - com.typesafe.akka.slf4j - 2.3.10 | Remoting started;
listening on addresses :[akka.tcp://opendaylight-cluster-data@192.168.56.101:2550]
2016-01-19 18:04:11,480 | INFO  | ult-dispatcher-2 | kka://opendaylight-cluster-data) | 266 - com.typesafe.akka.slf4j - 2.3.10
| Cluster Node [akka.tcp://opendaylight-cluster-data@192.168.56.101:2550] - Starting up...
2016-01-19 18:04:11,625 | INFO  | ult-dispatcher-3 | kka://opendaylight-cluster-data) | 266 - com.typesafe.akka.slf4j - 2.3.10
| Cluster Node [akka.tcp://opendaylight-cluster-data@192.168.56.101:2550] - Registered cluster JMX MBean
[akka:type=Cluster]
2016-01-19 18:04:11,626 | INFO  | ult-dispatcher-3 | kka://opendaylight-cluster-data) | 266 - com.typesafe.akka.slf4j - 2.3.10
| Cluster Node [akka.tcp://opendaylight-cluster-data@192.168.56.101:2550] - Started up successfully
```

Once Node 1 is successfully initialized, the Akka Clustering module starts working. It sends join messages to all the seed node configured in the configuration files. At this point as it is the only node initialized it won't get any answer back, this can be evidenced in the log, where is possible to see node 1 contacting other nodes but with not answer. The connection is refused as is shown in the following.

```
2016-01-19 18:04:12,321 | WARN  | ult-dispatcher-4 | ReliableDeliverySupervisor | 266 - com.typesafe.akka.slf4j - 2.3.10 |
Association with remote system [akka.tcp://opendaylight-cluster-data@192.168.56.102:2550] has failed, address is now
gated for [5000] ms. Reason: [Association failed with [akka.tcp://opendaylight-cluster-data@192.168.56.102:2550]] Caused
by: [Connection refused: /192.168.56.102:2550]
2016-01-19 18:04:12,343 | INFO  | ult-dispatcher-4 | rovider$RemoteDeadLetterActorRef | 266 - com.typesafe.akka.slf4j -
2.3.10 | Message [akka.cluster.InternalClusterAction$InitJoin$] from Actor[akka://opendaylight-cluster-
data/system/cluster/core/daemon/firstSeedNodeProcess-1#645930444] to Actor[akka://opendaylight-cluster-
data/deadLetters] was not delivered. [1] dead letters encountered. This logging can be turned off or adjusted with
configuration settings 'akka.log-dead-letters' and 'akka.log-dead-letters-during-shutdown'.
```

Having this in mind the next move is to join itself by sending a "Join message" to its direction and specifying its configured role “member - 1”. After that the node is set as up and the operation can start, this is possible only if it is the first node in the first node list.

```
2016-01-19 18:04:17,300 | INFO  | lt-dispatcher-19 | kka://opendaylight-cluster-data) | 266 - com.typesafe.akka.slf4j - 2.3.10
| Cluster Node [akka.tcp://opendaylight-cluster-data@192.168.56.101:2550] - Node [akka.tcp://opendaylight-cluster-
data@192.168.56.101:2550] is JOINING, roles [member-1]
2016-01-19 18:04:17,681 | INFO  | config-pusher    | StatisticsManagerModule   | 241 - org.opendaylight.controller
md.statistics-manager - 1.1.4.Helium-SR4 | StatisticsManager module initialization.
2016-01-19 18:04:17,729 | INFO  | ult-dispatcher-5 | kka://opendaylight-cluster-data) | 266 - com.typesafe.akka.slf4j - 2.3.10
| Cluster Node [akka.tcp://opendaylight-cluster-data@192.168.56.101:2550] - Leader is moving node
[akka.tcp://opendaylight-cluster-data@192.168.56.101:2550] to [Up]
```

In general when a node is started, it goes to memory to see if there is a module stored, if there is one it goes and read it from disk. These modules depends on which features are installed in the controller. In the case of clustering, the modules are the ShardManager-Operational and the ShardManager-Config.

```
2016-01-19 18:04:11,723 | INFO  | config-pusher    | DistributedDataStore           | 280 - org.opendaylight.controller.sal-
distributed-datastore - 1.1.4.Helium-SR4 | module shards config file exists - reading config from it
```

```
2016-01-19 18:04:11,754 | INFO  | config-pusher    | DistributedDataStore           | 280 - org.opendaylight.controller.sal-
distributed-datastore - 1.1.4.Helium-SR4 | modules config file exists - reading config from it
2016-01-19 18:04:12,226 | INFO  | config-pusher    | DistributedDataStore           | 280 - org.opendaylight.controller.sal-
distributed-datastore - 1.1.4.Helium-SR4 | module shards config file exists - reading config from it
2016-01-19 18:04:12,241 | INFO  | config-pusher    | DistributedDataStore           | 280 - org.opendaylight.controller.sal-
distributed-datastore - 1.1.4.Helium-SR4 | modules config file exists - reading config from it
```

If there are still some modules missing in the system, then it will create them. Normally this always happen when a node is started for the first time. In this case it has to create the modules because it is the first time the controller runs.

```
2016-01-19 18:04:11,960 | INFO  | config-pusher    | DistributedDataStore           | 280 - org.opendaylight.controller.sal-
distributed-datastore - 1.1.4.Helium-SR4 | Creating ShardManager : shardmanager-operational
2016-01-19 18:04:12,263 | INFO  | config-pusher    | DistributedDataStore           | 280 - org.opendaylight.controller.sal-
distributed-datastore - 1.1.4.Helium-SR4 | Creating ShardManager : shardmanager-config
```

When the modules are fully operated, this means that the system have read it from disk or it created from zero then the system gives a notification in the Log as the following.

```
2016-01-19 18:04:13,544 | INFO  | lt-dispatcher-19 | ShardManager                  | 280 - org.opendaylight.controller.sal-
distributed-datastore - 1.1.4.Helium-SR4 | Recovery complete : shard-manager-config
2016-01-19 18:04:13,550 | INFO  | ult-dispatcher-4 | ShardManager                  | 280 - org.opendaylight.controller.sal-
distributed-datastore - 1.1.4.Helium-SR4 | Recovery complete : shard-manager-operational
```

Now that all the modules are running in the system, the next step is to go to the initial configuration files in order to read which are the shards who need to be created. In this case are only the shards: inventory, topology, toaster and, default.

In this log explanation is only mentioned the shard-inventory-operational, but the process is the same for all the shards. Once the system creates the shards it goes and put them into the "InMemoryDataTree".

```
**2016-01-19 18:04:14,208** | INFO  | lt-dispatcher-20 | Shard                    | 273 - org.opendaylight.controller.sal-akka-raft -
1.1.4.Helium-SR4 | Shard created : member-1-shard-inventory-operational persistent : true
**2016-01-19 18:04:14,209** | INFO  | lt-dispatcher-20 | InMemoryDataTree              | 151 - org.opendaylight.yangtools.yang-
data-impl - 0.6.6.Helium-SR4 | Attempting to install schema contexts
```

Now that the shards are in the "InMemoryDataTree" it’s possible to recover the information from the journal and after that, it gives a notification which tells that the shard in ready.

```
**2016-01-19 18:04:14,279** | INFO  | lt-dispatcher-20 | Shard                    | 273 - org.opendaylight.controller.sal-akka-raft -
1.1.4.Helium-SR4 | member-1-shard-inventory-operational: Starting recovery with journal batch size 1
**2016-01-19 18:04:14,281** | INFO  | lt-dispatcher-20 | RoleChangeNotifier             | 272 - org.opendaylight.controller.sal-
clustering-commons - 1.1.4.Helium-SR4 | RoleChangeNotifier:akka.tcp://opendaylight-cluster-
data@192.168.56.101:2550/user/shardmanager-operational/member-1-shard-inventory-operational/member-1-shard-
inventory-operational-notifier#-1968314382 created and ready for shard:member-1-shard-inventory-operational
```

When a shard is ready to operate it always starts in the Follower state as is defined in the Raft algorithm. In the Log is possible to see how the node switchs the state of the shard to follower. After this step the system starts to perform the leader election process.

```
**2016-01-19 18:04:14,350** | INFO  | lt-dispatcher-19 | Shard                    | 273 - org.opendaylight.controller.sal-akka-raft -
1.1.4.Helium-SR4 | Recovery completed - Switching actor to Follower - Persistence Id =  member-1-shard-inventory-
operational Last index in log=-1, snapshotIndex=-1, snapshotTerm=-1, journal-size=0
```

Node 2

In this node is also observed the same procedure than in node 1. It starts and immediately try to communicate with seed nodes, in this case as node 1 is already in operation and it is the first one in the seed node list, there must be an answer for it. So the communication between both nodes is the following. First, node 2 sends a "Join message" to node 1 and it replies with a "Welcome message". This automatically creates a connection and allows to continue with the cluster deployment.

```
2016-01-19 18:04:59,926 | INFO  | lt-dispatcher-19 | kka://opendaylight-cluster-data) | 266 - com.typesafe.akka.slf4j - 2.3.10
| Cluster Node [akka.tcp://opendaylight-cluster-data@192.168.56.101:2550] - Node [akka.tcp://opendaylight-cluster-
data@192.168.56.102:2550] is JOINING, roles [member-2]
2016-01-19 18:05:01,869 | INFO  | lt-dispatcher-15 | kka://opendaylight-cluster-data) | 266 - com.typesafe.akka.slf4j - 2.3.10
| Cluster Node [akka.tcp://opendaylight-cluster-data@192.168.56.102:2550] - Welcome from [akka.tcp://opendaylight-cluster-
data@192.168.56.101:2550]
2016-01-19 18:05:00,753 | INFO  | ult-dispatcher-2 | kka://opendaylight-cluster-data) | 266 - com.typesafe.akka.slf4j - 2.3.10
| Cluster Node [akka.tcp://opendaylight-cluster-data@192.168.56.101:2550] - Leader is moving node
[akka.tcp://opendaylight-cluster-data@192.168.56.102:2550] to [Up]
```

When all the nodes in the cluster are up and the shards have created in each node then it’s time to start with the leader election for each shard, this is done with the implementation of the Raft algorithm.

As was explained before, all the shards start as followers and wait for a "heartbeat" in a determinate time, it they don’t receive anything they can change their status to candidate and try to obtain votes from other nodes. So they send request vote messages and if the majority vote for a node, that node can changes its status to leader.

In the log is possible to observe when a shard change its status from follower to candidate after waiting for a determinate time “timeout”. In this case is the shard of node 1 which changes the status to candidate.

```
2016-01-19 18:04:24,425 | INFO  | lt-dispatcher-20 | Shard                            | 273 - org.opendaylight.controller.sal-akka-raft -
1.1.4.Helium-SR4 | member-1-shard-inventory-operational (Follower) :- Switching from behavior Follower to Candidate
2016-01-19 18:04:24,425 | INFO  | lt-dispatcher-20 | RoleChangeNotifier             | 272 - org.opendaylight.controller.sal-
clustering-commons - 1.1.4.Helium-SR4 | RoleChangeNotifier for member-1-shard-inventory-operational , received role
change from Follower to Candidate
```

Once the votes are received, it is possible to set a leader for a determinate shard as following.

```
2016-01-19 18:05:05,241 | INFO  | ult-dispatcher-3 | Shard                            | 273 -
org.opendaylight.controller.sal-akka-raft - 1.1.4.Helium-SR4 | member-1-shard-inventory-operational
(Candidate) :- Switching from behavior Candidate to Leader
2016-01-19 18:05:05,244 | INFO  | ult-dispatcher-5 | RoleChangeNotifier               | 272 -
org.opendaylight.controller.sal-clustering-commons - 1.1.4.Helium-SR4 | RoleChangeNotifier for member-1-
shard-inventory-operational , received role change from Candidate to Leader
```

The part above described is very important because doing this analysis is the only way to actually understand what is really happening in the system internally. And this will give us also an idea of how to solve problems.

11. Messages between Controllers

It was already explained how the system works internally, but still there has to be explained how the nodes are interacting each other. Interacting means the actual data exchange between controllers. Having that in mind, is performed a packet data capture in order to determine how is the structure of an OpenDayLight packet and also the different messages that are implemented.

This is going to be one of the most important parts of the project because it will set some parameters for future interactions with different controllers, having the tools for building a proxy between the OpenDayLight controller and another kind of controller. This will allow for future clustering architectures in which the cluster can be compose for different kind of controllers working together.

For this packet capture was used Wireshark [20] that is a packet analyzer tool, with these tool was possible to filter all the packets needed and also to separate only the desired information contained in the payload.

11.1.Capture

Once the controllers start, they try to communicate with seed nodes as was explained before, first by establishing a TCP channel. If the node that is being contacted is alive then it will allow the connection. Otherwise the connection will be refused. This TCP channel is built with the three handshake mechanism.

During the first part of the clustering the Akka tool is in charge of all the communication and then the Raft algorithm takes place.

11.1.1. Akka tool Part

After the handshake and with the channel established, nodes try to join seed nodes by sending a message, the message basically contains its direction as is shown in the following.

```
...D.B...>3.opendaylight-cluster-data..192.168.56.102...".tcp.........
```

After the first contact, the joining node has to wait until any of the seed nodes answer to that first contact. This answer is very simple, it contains its address.

```
...D.B...>3.opendaylight-cluster-data..192.168.56.101...".tcp.].......
```

When the first contact is done, the joining node can start with the joining process. This is done by initiating an internal action call "Join Action". This process also has an identifier number that is included in the message. The message is like the following.

```
...%.....;9akka.tcp://opendaylight-cluster-
data@192.168.56.101:2550/.gc........system......cluster......core.....daemon",akka.cluster.InternalClusterAction$InitJoin$(..."w
uakka.tcp://opendaylight-cluster-data@192.168.56.102:2550/system/cluster/core/daemon/joinSeedNodeProcess-1#-
2021912680
```

Once the seed node has received the previous initialize message, it sends back an acknowledgment of that join process in order to continue with the joining. The whole communication is performed with the help of the identifier number, with that number is possible to identify different processes.

```
uakka.tcp://opendaylight-cluster-data@192.168.56.102:2550/system/cluster/core/daemon/joinSeedNodeProcess-1#-
2021912680.l8.opendaylight-cluster-data..192.168.56.101...".akka.tcp....akka.cluster.InternalClusterAction$InitJoinAck"`
^akka.tcp://opendaylight-cluster-data@192.168.56.101:2550/system/cluster/core/daemon#1239809082
```

After the acknowledgment, the joining node has to send information about its configuration (member or role in the cluster). This is done because there are some configurations in which each member have a different role. In this case the role defined in the initial configuration is “member-2”.

```
...V.....;9akka.tcp://opendaylight-cluster-data@192.168.56.101:2550/.....K?8.opendaylight-cluster-data..192.168.56.102..."
.akka.tcp....v..member-2.......system......cluster......core.....daemon"'akka.cluster.InternalClusterAction$Join(..."`
^akka.tcp://opendaylight-cluster-data@192.168.56.102:2550/system/cluster/core/daemon#2118301878
```

Now that the seed node is aware of the role of the joining node, it can finish the initiation process by changing the status of the joining node to up. It also has to inform the joining node that it is up, this is done with a welcome message.

```
^akka.tcp://opendaylight-cluster-data@192.168.56.102:2550/system/cluster/core/daemon#2118301878................r...../
H.KI...L.(.M.)-.I-.MI,I..3.4.34..35.340..&.......W.\ p....|B...(6......PM....L.#757.....2.R00M206JJMKL.L25.HJ.01I2.4M5LLL3KU
..`.`.`TbdP..`.......J..ZL..F...B,@.......&......*akka.cluster.InternalClusterAction$Welcome"`^akka.tcp://opendaylight-cluster -
data@192.168.56.101:2550/system/cluster/core/daemon#1239809082
```

From this point on, the joining node starts to be part of the cluster. This procedure is the same for all the nodes who want to join the cluster. After the joining process all the members in the cluster can start to exchange information, this is achieved with the help of the Akka clustering module.

There are still 2 more messages that are part of this module, one is the heartbeat and the other is the gossip message.

Heartbeat is a really important message it’s used in order to maintain the up state of the members in the cluster, that’s why it’s repeated during the whole operation of the cluster. If a heartbeat message is not answered it means that some node is not reachable and the system marks it like that. That node stays as a part of the cluster as far as it does not reach the **Phi accrual fail detector** threshold, otherwise it has to be removed from the system.

A heartbeat message also has integrated a heartbeat sender number and it is useful to identify different actors. It looks like the following.

```
9akka.tcp://opendaylight-cluster-data@192.168.56.102:2550/.....8.opendaylight-cluster-data..192.168.56.101...".akka.tcp...
....system......cluster......heartbeatReceiver"-akka.cluster.ClusterHeartbeatSender$Heartbeat(..."qoakka.tcp://opendaylight -
cluster-data@192.168.56.101:2550/system/cluster/core/daemon/heartbeatSender#-1807478239
```

Every heartbeat message has to be answered or acknowledged, otherwise the node will reach the value of the phi accrual fail detector. A heartbeat answer message looks like the following

```
...Y.....qoakka.tcp://opendaylight-cluster-data@192.168.56.101:2550/system/cluster/core/daemon/heartbeatSender#-180747
8239.u?8.opendaylight-cluster-data..192.168.56.102...".akka.tcp....v...0akka.cluster.ClusterHeartbeatSender $HeartbeatRsp
"geakka.tcp://opendaylight-cluster-data@192.168.56.102:2550/system/cluster/heartbeatReceiver#-1251842346
```

Gossip message are also really important for the system, inside this message there is useful information about the cluster as the state and versioning. This is the key to try to build a convergence system, gossip messages are exchanged every second and, they are random in the sense that they don't have a fixed destination. The message is like the following.

```
9akka.tcp://opendaylight-cluster-data@192.168.56.102:2550/.......@8.opendaylight-cluster-data..192.168.56.101...".akka.tcp
....?8.opendaylight-cluster-data..192.168.56.102...".akka.tcp....v.............r...../H.KI...L.(.M.)-.I-.MI,I..3.4.34..35.340..&.......W.\
p....|\...`.j....eB....I@..p.....i...QRjZb.e...ER...I...i.abb.Y..'.......#../.... `.`Pbd.b.`0..`.b..`.....T...........system......cluster......core.....da
emon".akka.cluster.GossipEnvelope(..."`^akka.tcp://opendaylight-cluster-data@192.168.56.101:2550/system/cluster/core/
daemon #1239809082
```

The gossip message also has to be acknowledged as the following

```
...,.....`^akka.tcp://opendaylight-cluster-data@192.168.56.101:2550/system/cluster/core/daemon#1239809082.....?8.open
daylight-cluster-data..192.168.56.102...".akka.tcp....v.@8.opendaylight-cluster-data..192.168.56.101...".akka.tcp...............
....r...../H.KI...L.(.M.)-.I-.MI,I..3.4.34..35.340..&.......W.\p....|\...`.j....eB....I@..p.....i...QRjZb.e...ER...I...i.abb.Y..'.......#../....
`.`Pbd.b.`.`4..`........>.........akka.cluster.GossipEnvelope"`^akka.tcp://opendaylight-cluster-data@192.168.56.102:2550
/system/cluster/core/daemon#2118301878
```

Up to here acts the Akka clustering tool, and from now on the principal role will be under the supervision of the raft algorithm.

11.1.2. Raft algorithm part

When the clustering services start, it is time to begin with the leader election for all the shards implemented in the system. As was explained before in Raft algorithm description, the first node that times out will become a candidate and will send request votes. This is with the aim of obtain the approbation of the majority of the nodes and change the status to leader of a given shard.

As before it will be shown only the case of the Shard-Inventory-Operational message as an example, but all the others are completely the same. The message brings information about the current term and candidate ID. The request vote message looks like the following.

```
9akka.tcp://opendaylight-cluster-data@192.168.56.102:2550/...........sr.=org.opendaylight.controller.cluster.raft.messages.
RequestVotem[}.a.)&...J..lastLogIndexJ..lastLogTermL..candidateIdt..Ljava/lang/String;xr.Aorg.opendaylight.controller.cluste
r.raft.messages.AbstractRaftRPC......l ...J..termxp.......................t.$member-1-shard-inventory-operational........user......sha
rdmanager-operational.(...$member-2-shard-inventory-operational(...."....akka.tcp://opendaylight-clusterdata@192.168.56.101
:2550/user/shardmanager-operational/member-1-shard-inventory-operational#-1784895488
```

The answer for this messages is really similar and in addition it brings information about if the vote was granted or not. The message looks like the following.

```
.............akka.tcp://opendaylight-cluster-data@192.168.56.101:2550/user/shardmanager-operational/member-1-shard-
inventory-operational#-1784895488.........sr.Borg.opendaylight.controller.cluster.raft.messages.RequestVoteReply
..._.......Z..voteGrantedxr.Aorg.opendaylight.controller.cluster.raft.messages.AbstractRaftRPC......l ...J..termxp..........."..
..akka.tcp://opendaylight-cluster-data@192.168.56.102:2550/user/shardmanager-operational/member-2-shard-inventory-
operational#-870461665
```

Once a candidate node receives the majority of the votes, it will become the leader of a given shard during a given term. After that moment it starts to replicate data to the other nodes. In this case of Akka, that kind of message is called “Append Entries" message and, it can also be used as a way of heartbeat message. This is with the aim of establish if a given leader is up or down, if the leader is down there has to be another election turn. A typical append entries message is like the following.

```
9akka.tcp://opendaylight-cluster-data@192.168.56.102:2550/.....T...$member-1-shard-inventory-operational...........
..........0..........8.................user......shardmanager-operational.(...$member-2-shard-inventory-operational"_org.opendaylight
.controller.protobuff.messages.cluster.raft.AppendEntriesMessages$AppendEntries(..."....akka.tcp://opendaylight-cluster-
data@192.168.56.101:2550/user/shardmanager-operational/member-1-shard-inventory-operational#-1784895488
```

The answer to this message is the following.

```
...`.........akka.tcp://opendaylight-cluster-data@192.168.56.101:2550/user/shardmanager-operational/member-1-shard-
inventory-operational#-1784895488.........sr.Dorg.opendaylight.controller.cluster.raft.messages.AppendEntriesReply...Y.N.
....J..logLastIndexJ..logLastTermZ..successL.followerIdt..Ljava/lang/String;xr.Aorg.opendaylight.controller.cluster.raft.messag
es .AbstractRaftRPC......l ...J..termxp........................t.$member-2-shard-inventory-operational.."....akka.tcp://opendaylight-
cluster-data @192.168.56.102:2550/user/shardmanager-operational/member-2-shard-inventory-operational#-870461665
```

In this case is very easy to note that the "Append Entries" message is empty, this is because when there is not information to exchange the message is used as a heartbeat message.

In this work was also done a different test, in which a small network simulated in Mininet was connected to the cluster of controllers. This was made with the aim of put some information about inventory and topology to the controller. As a result the system creates a transaction and starts to replicate all the data from the shard leaders to the followers. An example of that message will be the following.

```
9akka.tcp://opendaylight-cluster-data@192.168.56.102:2550/..........#member-1-shard-topology-operational.. .*.........iorg.
opendaylight.controller.cluster.raft.protobuff.client.messages.CompositeModificationByteStringPayload.....Rclass org.open
daylight.controller.cluster.datastore.modification.MergeModification........ .0..Y....... .0. .b+urn:TBD:params:xml:ns:yang:
network-topologyb2013-10-21b.network-topology..Rclass org.opendaylight.controller.cluster.datastore.modification.Merge
Modification........ .0....... .0..c....... .0. .b+urn:TBD:params:xml:ns:yang:network-topologyb2013-10-21b.topologyb. network-
topology..Rclass org.opendaylight.controller.cluster.datastore.modification.WriteModification.8...... .0....... .0........  "......
...flow:1..0............ ."...... ...flow:1..0. .2........ .0. .:.flow:1H.b+urn:TBD:params:xml:ns:yang:network-topologyb2013-10-
21b.topologyb.topology-idb.network-topology......50.8.................user......shardmanager-operational.'...#member-2 -shard-
topology-operational"_org.opendaylight.controller.protobuff.messages.cluster.raft.AppendEntries Messages$AppendEntries
(..."....akka.tcp://opendaylight-cluster-data@192.168.56.101:2550/user/shardmanager-operational/member-1-shard-topology-
operational#-1971300728
```

In response to the above message, follower's replies with a “Create Transaction” message and send it to the leader. The message is the following

```
...c.....;9akka.tcp://opendaylight-cluster-data@192.168.56.101:2550/.......member-2-txn-0.... .........user......shardmanager-
operational.'...#member-1-shard-topology-operational"eorg.opendaylight.controller.protobuff.messages.transaction.
ShardTransactionMessages$CreateTransaction(..."B@akka.tcp://opendaylight-cluster-data@192.168.56.102:2550/temp/$a
```

Then the leader answers with an acknowledgement like the following.

```
@akka.tcp://opendaylight-cluster-data@192.168.56.102:2550/temp/$a.......akka.tcp://opendaylight-cluster-data@192.168
.56.101:2550/user/shardmanager-operational/member-1-shard-topology-operational/shard-member-2-txn-0#1175605301
..member-2-txn-0.....jorg.opendaylight.controller.protobuff.messages.transaction.ShardTransactionMessages$
CreateTransactionReply"....akka.tcp://opendaylight-cluster-data@192.168.56.101:2550/user/shardmanager-operational
/member-1-shard-topology-operational#-1971300728
```

The above mentioned is due to a modification in a given shard, so there has to be a transaction creation that is generated for a given actor. After the creation of the transaction and once the followers have received all the information then they replies to the leader with a message that means that the information received is going to be merged with the old information in the shard. The message is like the following.

```
........;9akka.tcp://opendaylight-cluster-data@192.168.56.101:2550/.....i....... .0..Y....... .0. .b+urn:TBD:params:xml:ns:yang
:network-topologyb2013-10-21b.network-topology........user......shardmanager-operational.'...#member-1-shard-topology-
operational.#....shard-member-2-txn-0#1175605301"]org.opendaylight.controller.protobuff.messages.transaction.Shard
TransactionMessages$MergeData(..."B@akka.tcp://opendaylight-cluster-data@192.168.56.102:2550/temp/$b
```

Then the leader reply with an acknowledgment of that merge operation, as the following.

```
@akka.tcp://opendaylight-cluster-data@192.168.56.102:2550/temp/$b.h....borg.opendaylight.controller.protobuff
.messages.transaction.ShardTransactionMessages$MergeDataReply"....akka.tcp://opendaylight-cluster-data@192
.168.56.101:2550/user/shardmanager-operational/member-1-shard-topology-operational/shard-member-2-txn-
0#1175605301
```

When the followers have the merged information, now is possible to write it in memory. They have to tell the leader that they will apply all the changes in the shard to memory. The message is the following.

```
...F.....;9akka.tcp://opendaylight-cluster-data@192.168.56.101:2550/.......8...... .0....... .0........ ."...... ...flow:1..0............ .".
..... ...flow:1..0. .2........ .0. .:.flow:1H.b+urn:TBD:params:xml:ns:yang:network-topologyb2013-10-21b.topologyb. topology-
idb.network-topology........user......shardmanager-operational.'...#member-1-shard-topology-operational.#....shard-member-2-
txn-0#1175605301"]org.opendaylight.controller.protobuff.messages.transaction.ShardTransactionMessages$WriteData(..."B
@akka.tcp://opendaylight-cluster-data@192.168.56.102:2550/temp/$d
```

It also has to be acknowledged for the leader and, the message is the following.

```
@akka.tcp://opendaylight-cluster-data@192.168.56.102:2550/temp/$d.h....borg.opendaylight.controller.protobuff
.messages.transaction.ShardTransactionMessages$WriteDataReply"....akka.tcp://opendaylight-cluster-data@192. 168.56
.101:2550/user/shardmanager-operational/member-1-shard-topology-operational/shard-member-2-txn-0#1175605301
```

Once all the followers have the information in memory they notify the leader that the information can be committed into its data tree. The message is like the following.

```
...p.....;9akka.tcp://opendaylight-cluster-data@192.168.56.101:2550/.......member-2-txn-0........user......shardmanager-
operational.3.../member-1-shard-topology-operational#-971300728"lorg.opendaylight.controller.protobuff.messages.c
ohort3pc.ThreePhaseCommitCohortMessages$CanCommitTransaction(..."B@akka.tcp://opendaylight-cluster-
data@192.168.56.102:2550/temp/$f
```

The leader sends back an acknowledgment, as the following.

```
@akka.tcp://opendaylight-cluster-data@192.168.56.102:2550/temp/$f.y......qorg.opendaylight.controller.protobuff.messages
.cohort3pc.ThreePhaseCommitCohortMessages$CanCommitTransactionReply"....akka.tcp://opendaylight-cluster-
data@192. 168.56.101:2550/user/shardmanager-operational/member-1-shard-topology-operational#-1971300728
```

Finally the leader can have an idea of which of the followers have a given information. If that number of followers is the majority, it goes into its data tree and commits the information. After this point that information will never be overwritten, that means that will be durable and persistent.

12. Bandwidth Usage Analysis

This chapter contains objective of the thesis, the evaluation of the traffic exchanged between controllers or “east-west traffic”. In this work was implemented a cluster topology in which was also implemented a Mininet network [21], this virtual network was varied in size with the aim of analyze the bandwidth usage for the network under different situations. Here will be also described the methodology of experimentation and all the steps during the analysis.

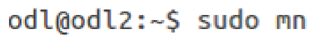
12.1. Experimentation Methodology

The methodology for this work will be the following. For this part was performed a data packet capture, was used a network protocol analyzer, in this case "Wireshark" [20]. Then with the data obtained was performed a data processing in order to obtain useful information and graphs of the bandwidth, for this process was used Matlab [22].

12.2. Mininet

As described in [21], Mininet is a network emulator which creates a network of virtual hosts, switches, controllers, and links. Mininet hosts run standard Linux network software, and its switches support OpenFlow for highly flexible custom routing and Software-Defined Networking. It creates a realistic virtual network, running real kernel, switch and application code, on a single machine (VM, cloud or native), in seconds, with a single command. It supports research, development, learning, prototyping, testing, debugging, and any other tasks that could benefit from having a complete experimental network on a laptop or other PC.

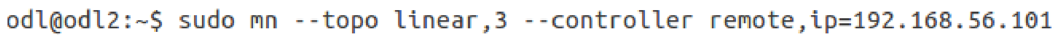
Mininet for default brings its own controller, but there is also the possibility to use a remote one. This is exactly the kind of network emulator needed for the experimentation, the Mininet network will be connected to a SDN controller "OpenDayLight" through a TCP/IP connection. The simplest network in Mininet is composed for a single switch and 2 hosts and, it can be executed with the following command.



```
odl@odl2:~$ sudo mn
```

There are also some other configurations pre-established with a single line of command like: linear, single and tree topology, but there is also the possibility to create custom topologies with help of python scripts.

In this case it will be used the linear topology that can be run in the following way.



```
odl@odl2:~$ sudo mn --topo linear,3 --controller remote,ip=192.168.56.101
```

The command above has always to start with "mn" along with some parameters to define the kind of topology and controller to be use. Here it will be used only the remote controller, in order to test the OpenDayLight controller.

In the linear topology all the switches are connected in a linear fashion and each switch has its own host as is shown in Figure 12-1.

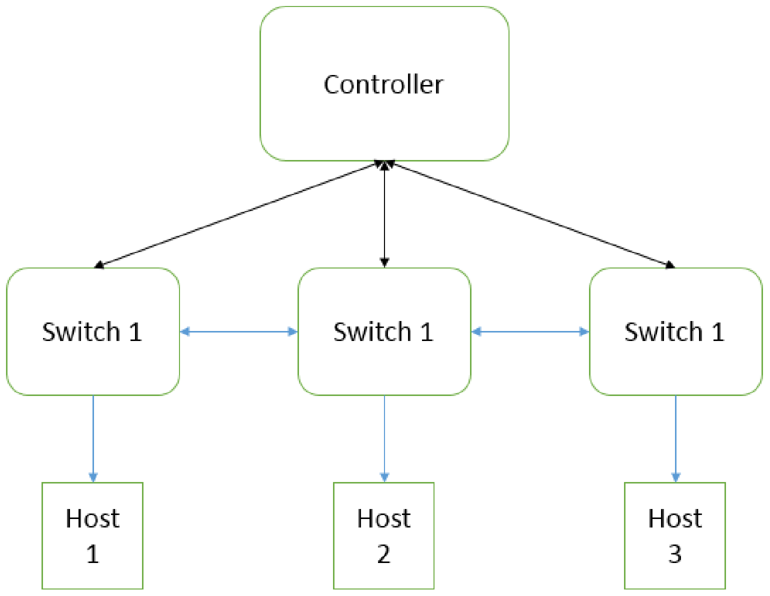
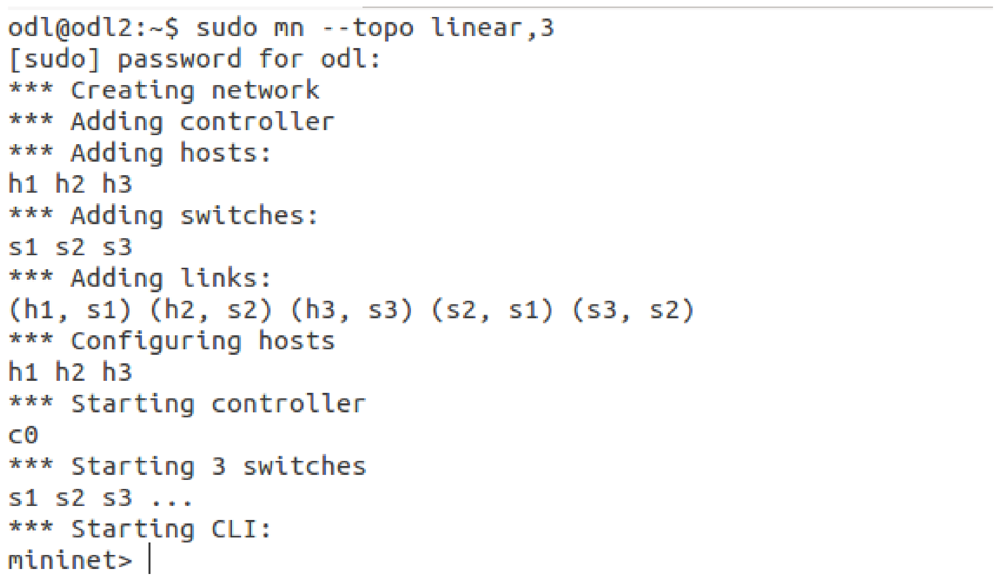


Figure 12-1 – Linear Topology

After running the command for the network creation, it will be shown how is been created and right after the system will open a Mininet terminal as is shown in Figure 12-2 in which can be executed some command related with the network.



```
odl@odl2:~$ sudo mn --topo linear,3
[sudo] password for odl:
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1 s2 s3
*** Adding links:
(h1, s1) (h2, s2) (h3, s3) (s2, s1) (s3, s2)
*** Configuring hosts
h1 h2 h3
*** Starting controller
c0
*** Starting 3 switches
s1 s2 s3 ...
*** Starting CLI:
mininet>
```

Figure 12-2 - Mininet terminal

In this terminal is possible to perform connectivity tests, pinging and also get information about the network like: nodes, links and addresses.

12.3.Data capture and processing

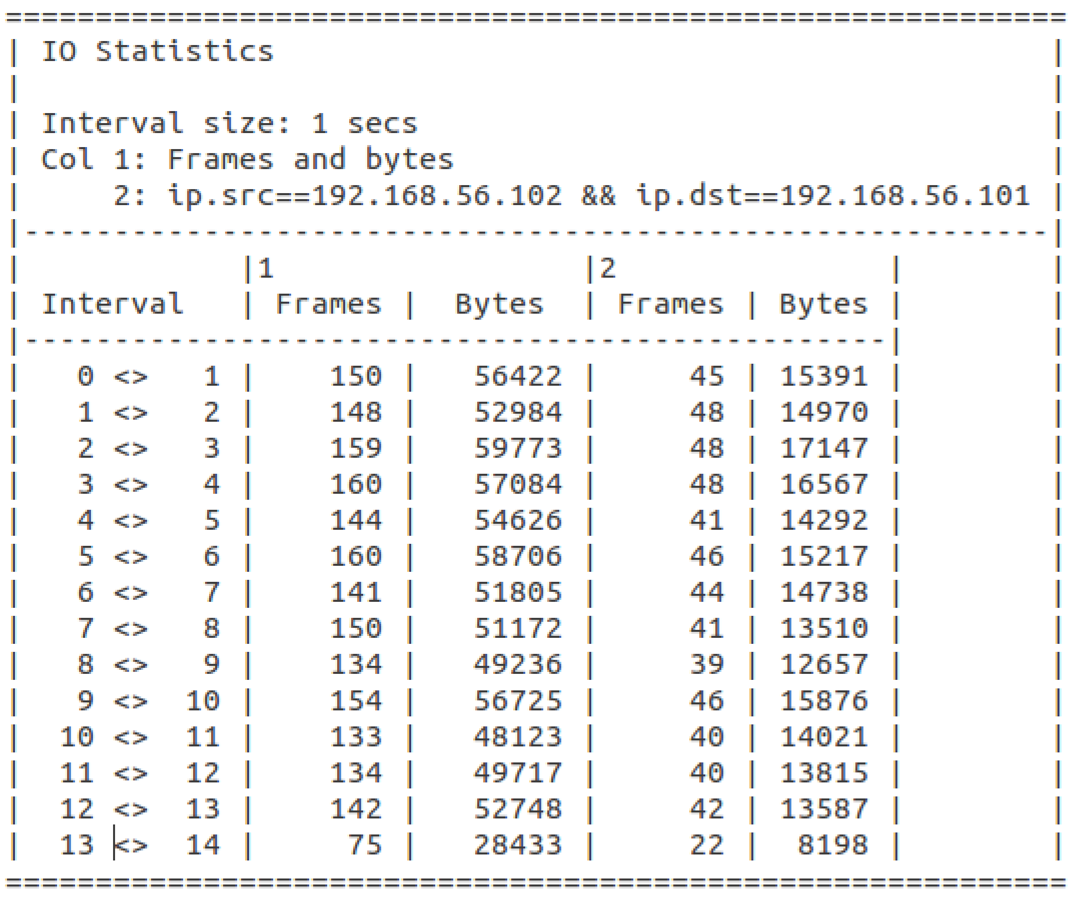
In this section will be described how was captured and processed all the information exchanged between OpenDayLight controllers, this information will be used also in the last part of the work with the aim of estimate an experimental model for the bandwidth usage in a cluster architecture.

During the whole work, the cluster architecture was composed for three nodes or instances of the OpenDayLight controller. For simplicity purpose in this part will be analyzed the traffic between only two of them, it will be named node1 as controller A and, node2 as controller B. with these two nodes is going to be performed the data capture by using Wireshark tool. This traffic will be captured in both directions from A → B and, from b → A.

The Wireshark capture contains a lot of information, but for obvious reasons in this part it will focus in the amount of kbps transferred between the controllers A and B. with these information it will be possible to make further estimations and analysis, unfortunately Wireshark does not have an exportation format that can be directly used for Matlab. That's why some Linux commands are going to be used to export the data needed from the Wireshark file, basically it is filtered all the original information and then the important one is selected. The above mentioned is performed with the following command.

```
tshark -q -nr source_file.pcapng -z "io,stat,0.1,ip.src==10.0.1.101 && ip.dst==10.0.1.102" | tail -n +13
| head -n -1 | awk '{print $2,$12}' > result.txt
```

The above command filters the source file and then it uses a sample frequency in order to obtain a matrix with some data that can be differentiated for time, from this matrix are used the Linux commands as tail, head and, awk as is shown in Figure 12-3.



```
=============================================================
| IO Statistics                                             |
|                                                           |
| Interval size: 1 secs                                     |
| Col 1: Frames and bytes                                   |
|     2: ip.src==192.168.56.102 && ip.dst==192.168.56.101 |
|-----------------------------------------------------------|
|             |1                  |2                |       |
| Interval    | Frames |  Bytes   | Frames | Bytes |        |
|-------------------------------------------------|         |
|   0 <>   1 |    150 |   56422 |      45 | 15391 |         |
|   1 <>   2 |    148 |   52984 |      48 | 14970 |         |
|   2 <>   3 |    159 |   59773 |      48 | 17147 |         |
|   3 <>   4 |    160 |   57084 |      48 | 16567 |         |
|   4 <>   5 |    144 |   54626 |      41 | 14292 |         |
|   5 <>   6 |    160 |   58706 |      46 | 15217 |         |
|   6 <>   7 |    141 |   51805 |      44 | 14738 |         |
|   7 <>   8 |    150 |   51172 |      41 | 13510 |         |
|   8 <>   9 |    134 |   49236 |      39 | 12657 |         |
|   9 <>  10 |    154 |   56725 |      46 | 15876 |         |
|  10 <>  11 |    133 |   48123 |      40 | 14021 |         |
|  11 <>  12 |    134 |   49717 |      40 | 13815 |         |
|  12 <>  13 |    142 |   52748 |      42 | 13587 |         |
|  13 <>  14 |     75 |   28433 |      22 |  8198 |         |
=============================================================
```

Figure 12-3 – Data matrix Capture

In this case is only selected the column of time and bytes. The last part of the command is used to save a file containing the information needed for the analysis, in this case the result file is a two columns file, containing time and bytes that can be subsequently loaded in Matlab.

For the experimentation, a Mininet network was connected to the cluster at the time of 60s and subsequently removed at 120s. After this part the data obtained will be processed in Matlab.

In order to graph the bandwidth usage in more suitable way, it’s needed to do some processing, for this thesis was used the **sliding window approach** [23]**.** This approach is basically take all the bytes exchanged between controllers and sum them and, then perform a normalization with the sample time that was implemented before.

As describe in [23] the sliding window approach is performed in the following way.

- A window size (W) is chosen with width more than the sampling interval.
- The window is centered on the starting sample of the signal and the mean of the signal values in the window is calculated and assigned to the center value.
- In the next iteration, the window moves one sample to the right and computes mean in the same way in the current window as in previous step and this is continued till the end of signal. Overlapping occurs between windows in each iteration.

In the end of the processing there will be a smoother signal than the original one, but still the data can be used to construct the model. In Figure 12-4 is shown the bandwidth usage for a Mininet network with size of 3 nodes, with different values of Ts and, W.

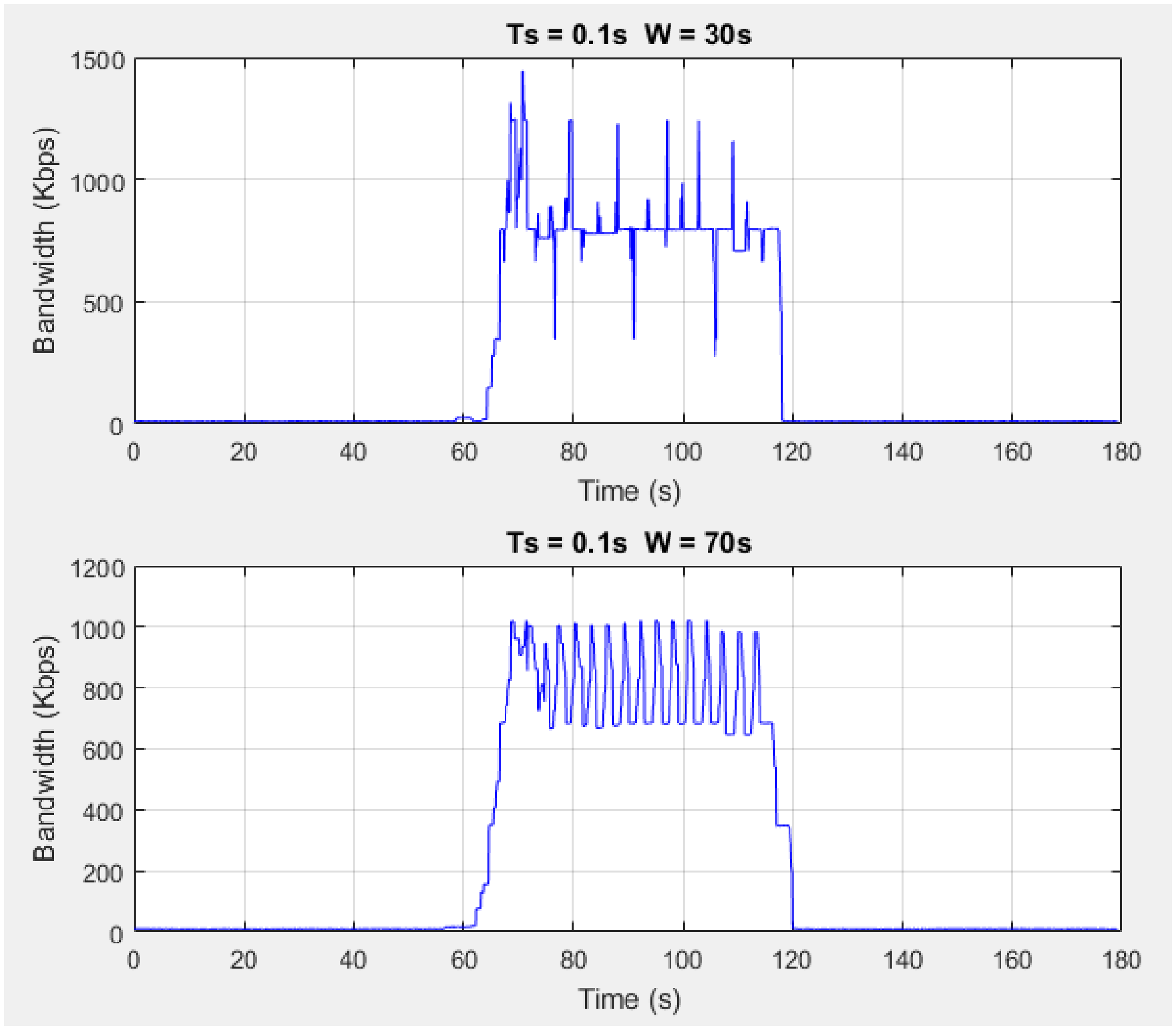


Figure 12-4 – Bandwidth Usage Graph A → B

12.4.Modeling

Once the capture it is finished is possible to use that data to build an experimental model for the traffic exchange between OpenDayLight controllers, this model can be used for the evaluation of scalability in SDN networks. With this model can be easily predicted different scenarios in a SDN network with the OpenDayLight controller, the model makes use of all the experimental captures in order to create an estimation for all the possible cases.

The procedure is the following, first it’s necessary to obtain the needed data. For this purpose a Mininet network was connected to the controllers, this Mininet network has a linear architecture and, it will vary in size.

It will be deployed a linear topology with sizes of: 1,3,5,7,9,12,15,20,25,30,40,50 switches. Unfortunately after 50 nodes our computational resources are not able to run in a properly way, that’s why the topology is only vary up to 50 nodes. However the fitting curve will be expanded up to 100 nodes. The set up for the modeling is shown in Figure 12-5.

During the experimentation the procedure was repeated 5 times for each topology size, this is with the aim of having a more accurate measurement. The process is the following.

1. Start all the controllers that are going to be part of the network.
2. Set the topology wanted and make the connection to the controller.
3. Start the packet capture during 180s, while the virtual network is connected to the controllers.
4. Stop the virtual network after the established time.
5. Stop the packet capture 60s after the stopping of the virtual network.
6. Shutdown all the controllers in the cluster.
7. Extract all the useful data from the packet capture, this is performed by using the Tshark tool. The information is extract in a .txt file. It contains 2 variables, one is the time and the other is the number of bits used.
8. Open the .txt file in Matlab for processing.

9. Generate the model and graphs.

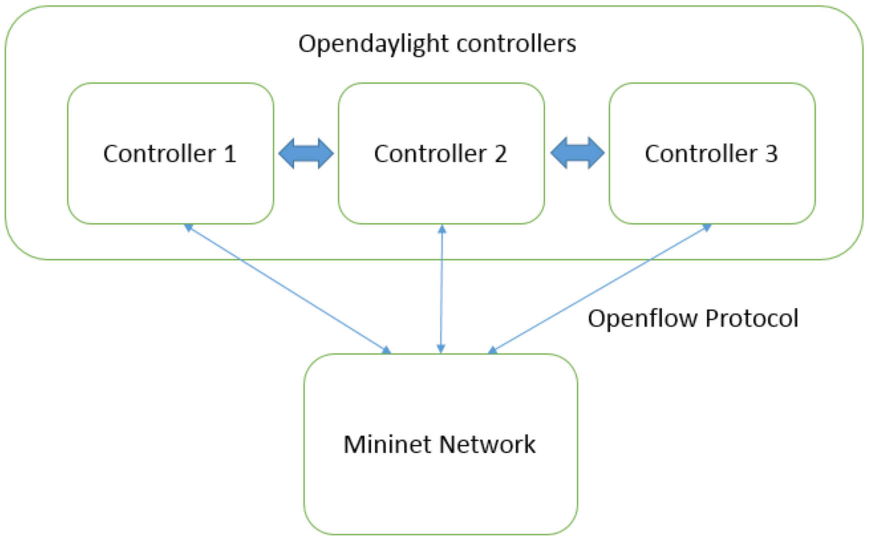


Figure 12-5 - Test setup for Modeling

The model is generated by using the amount of bytes transferred “Bandwidth” from A → B and, from B → A. there is also needed the topology size. These information is passed to a Matlab function called Polynomial curve fitting [24], which uses the least squares method to find a best fit curve. As a result Matlab gives a fitted curve on the mean values. Figure 12-6 shows the results for the model from A → B and, from B → A.

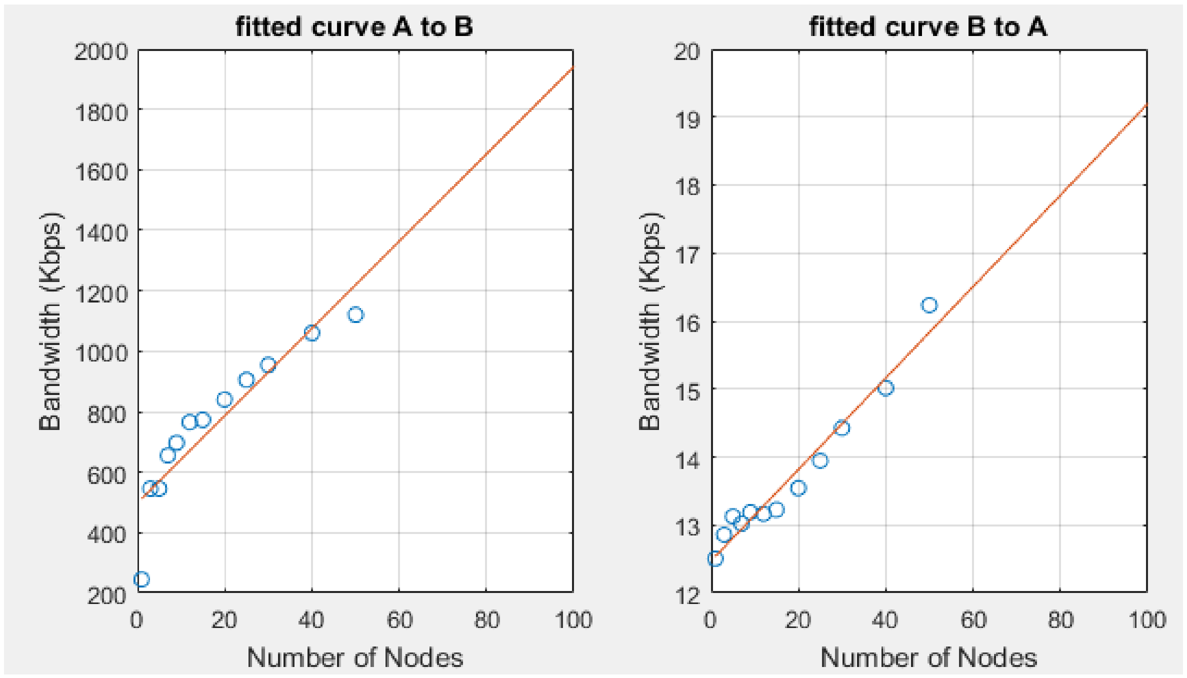


Figure 12-6 – Bandwidth Usage Modeling

In the previous graph it's possible to evidence the variation in bandwidth with respect to the number of nodes in the network.

In the first case from A → B, it is possible to observe that the bandwidth is increasing in a linear way with respect to the number of nodes.

For the second case from B → A, is also possible to observe the same linear increasing, but in this case consuming way less bandwidth that in the previous one.

The explanation to the behavior shown in the previous graph is because in the clustering with OpenDayLight the information only flows from leader to followers, that's why there is more bandwidth usage from A to B than from B to A. the leader has to send all the information about Topologies, Inventory and also the information related to the membership between controllers. Instead other nodes has to exchange information only about membership, that’s why it consume less bandwidth.

13. Conclusions

The idea of migrating traditional networks into Software Defined Networks is being supported for most of the greatest vendors and companies, therefore is needed a further research in the field of the SDN controllers in order to determine which one is the most efficient for the network. SDN networks have demonstrated that can be more flexible and reliable in shaping traffic along the network, making use of its decoupled architecture.

In this thesis was discussed the importance of having a distributed SDN network, in order to avoid the problem of having a single point of failure and also because with a distributed system is possible to achieve high availability, scalability and, persistency of the information. Was also analyzed the behavior of the OpenDayLight controller under the cluster architecture schema, with the aim of having a better understandability of how is acting the controller. One of the biggest part of the work was the research for the protocols that were being used for the network in the communication between controllers, in this part was also explained the different messages exchanged between controllers. This was performed with help of Wireshark, making traffic captures along the interfaces involved and analyzing the packets with the aim of understand the communication.

For the last part of the thesis was performed a bandwidth usage analysis by making a traffic capture between controllers. Was again implemented Wireshark for the data capture and then implemented Matlab for the data processing. In this chapter was implemented an experimental model for the bandwidth usage in different scenarios. The process consisted in vary the topology size of the network, this was implemented with a liner topology in Mininet and, then it was connected to the master controller.

With the model was possible to observe the behavior of the bandwidth usage with respect to the topology size, and then the final result was that the bandwidth grows in a linear fashion with the topology size.

13.1.Future work

In the seeking for a migration into SDN networks is really important to prove and test all the different controllers that are currently available in the market, because so far there is not any standard defined for SDN networks. For this was performed an analysis in the messages exchanged between ODL controllers, in order to set some parameters for the understandability of the system. This can be later used in future integrations with different controllers, building a proxy between the ODL controller and a different one. For that was also explain all the protocols used in the clustering architecture.

Another future work could be the implementation of the cluster in real equipment, this because the deployment was performed in a virtualized environment and it can lead to some changes in the real implementation. This would be the possibility to verify the result obtained and compare the bandwidth usage model in real networks.

References

[1] C. Technical Report, "Forecast and Methodology, 2014-2019 White Paper.," Cisco Visual Networking Index, 2015.

[2] O. N. Foundation, "Software Defined Networking: The New Norm for Networks," ONF, White Paper, 2012.

[3] F. M. V. R. P. E. V. C. E. R. S. A. S. U. Diego Kreutz, "Software-defined networking: A comprehensive survey," *IEEE,* 2015.

[4] L. Faughnan, "Software Defined Networking," TechEntral, 2013.

[5] i. s. &. i. group, "Software Defined Networking Primer + Deep Dive into Big Switch Networks," Technology Research, 2012.

[6] L. D. J. Q. H. Z. Ying Li, "Multiple controller management in software defined," IEEE, Symposium on Computer Applications and Communications, 2014.

[7] L. Foundation, "Opendaylight Platform Overview," 2016. [Online]. Available: https://www.opendaylight.org/platform-overview-beryllium.

[8] L. Foundation, "OpenDayLight Definition," 2016. [Online]. Available: https://www.opendaylight.org/.

[9] D. A. Liubov Efremova, "What is in OpenDaylight," 2015. [Online]. Available: https://www.mirantis.com/blog/whats-opendaylight/.

[10] J. Ogando, "Distributed control: another choice for multi-station loading systems," Plastics technology, 1995.

[11] U. o. P. Insup Lee, "Introduction Distributed Systems, Lecture Notes," 2014. [Online]. Available: http://www.cis.upenn.edu/~lee/00cse380/lectures/ln13-ds.ppt.

[12] J. O. Diego Ongaro, "In Search of an Understandable Consensus Algorithm," Stanford University, 2014.

[13] J. O. Diego Ongaro, "Raft Visualization," 2014. [Online]. Available: https://raft.github.io/.

[14] M. Raja, «MD-SAL Clustering Internals, Linux Foundation,» 2015. [En línea]. Available: http://events.linuxfoundation.org/sites/events/files/slides/MD-SAL%20Clustering%20Internals.pdf.

[15] Akka, "Akka Definition," 2011. [Online]. Available: http://akka.io/.

[16] Akka, "Cluster Specification Akka," 2011. [Online]. Available: http://doc.akka.io/docs/akka/snapshot/common/cluster.html#cluster.

[17] Ubuntu, "Ubuntu Documentation," [Online]. Available: http://www.ubuntu.com/.

[18] Oracle, "virtualbox Documentation," [Online]. Available: https://www.virtualbox.org/.

[19] L. Foundation, "OpenDayLight User Guide - Helium Release," 2015. [Online]. Available: https://www.opendaylight.org/software/release-archives.

[20] W. Foundation, "wireshark Documentarion".

[21] M. Org, "Mininet Walkthrough," [Online]. Available: http://mininet.org/.

[22] Mathworks, "Matlab Documentation," [Online]. Available: http://www.mathworks.com/.

[23] A. S. Muqaddas, "Evaluation of Distributed ONOS Controllers," Master Thesis - Politecnico di Torino, 2015.

[24] Mathwork, "Polynomial Curve Fitting," [Online]. Available: http://www.mathworks.com/help/matlab/math/polynomial-curve-fitting.html.